Australian Government
Department of Defence
Defence Science and
Technology Organisation

Processable English:
The Theory Behind the PENG System

Kerry Trentelman

Command, Control, Communications and Intelligence Division
Defence Science and Technology Organisation

DSTO-TR-2301

ABSTRACT

This report describes the theoretical underpinnings of the PENG system. Designed by Rolf Schwitter, Marc Tilbrook, et al. at the Centre for Language Technology at Macquarie University, the system incorporates a text editor where authors write text in a controlled language called PENG. A controlled language processor translates PENG text to first-order logic via a discourse representation structure. The resultant logical theory can then be checked for consistency and informativity, and may also be used for question-answering by third-party reasoning services.

Published by

Command, Control, Communications and Intelligence Division
DSTO Defence Science and Technology Organisation
PO Box 1500
Edinburgh South Australia 5111 Australia

Telephone: (08) 8259 5555
Fax: (08) 8259 6567

AR-014-554
June 2009

APPROVED FOR PUBLIC RELEASE

Processable English:
The Theory Behind the PENG System

Executive Summary

This report describes the theoretical underpinnings of the PENG system. Designed by Rolf Schwitter, Marc Tilbrook, et al. at the Centre for Language Technology at Macquarie University, the system incorporates a text editor where authors write text in a controlled language called PENG. Authors of PENG text do not need to remember the restrictions placed on the language, since the PENG text editor guides the writing process. It does this by indicating the possible sentence constructs allowable from the current input. A controlled language processor translates PENG text to a logical theory which can then be checked for consistency and informativity, and may also be used for question-answering by third-party reasoning services. Although PENG is still a prototype, and has a number of issues, the system shows potential as a useful tool.

Authors

Kerry Trentelman
Command,Control,Communications Intelligence Div

Kerry received a PhD in computer science from the Australian National University in 2006. Her thesis topic was the formal verification of Java programs. She joined the Intelligence Analysis Group at the DSTO in 2007 and now works in the area of information fusion. Her research interests include natural language processing, logics for knowledge representation and program specification, and theorem proving and automated reasoning.

Contents

1. Introduction

Today's intelligence analysts are finding themselves overloaded with information. Valuable information – sometimes implicit – is often buried amongst masses of irrelevant data. Heralding from unstructured sources such as natural language documents, email, audio, images and video, this information must be extracted, cross-checked for accuracy, analysed for significance, and disseminated appropriately. As part of the DSTO's C3ID Intelligence Analysis Discipline, we believe that automating aspects of this process offers a practical solution to the problem of information overload. We propose an intelligence information processing architecture which includes speech processing, language translation, information extraction, data-mining and estimative intelligence components, as well as a sixth 'information fusion' component. It is our intention that this component automatically fuses – albeit in an intelligent way – information derived from the extraction process with data from a structured knowledge base. This process will involve resolving, aggregating, integrating and abstracting information – using the methodologies of Knowledge Representation and Reasoning – into a single comprehensive description of an individual or event. From such fused information we hope to obtain improved estimation and prediction, data-mining, social network analysis, and semantic search and visualisation.

This report describes the theoretical underpinnings of an alternative approach to text processing. Of particular interest to the information fusion team, this approach completely bypasses the need for information extraction and heavily lends itself to the fusion process. The PENG system – designed by Rolf Schwitter with Marc Tilbrook, *et al.* at the Centre for Language Technology at Macquarie University – incorporates a text editor where authors write text in a controlled language called 'PENG'. PENG – the name derived from 'Processable ENGlish' – consists of a strict subset of English. Authors of PENG text do not need to remember the restrictions placed on the language, since the PENG text editor guides the writing process. It does this by indicating the possible sentence constructs allowable from the current input. A controlled language processor translates PENG text to first-order logic *via* a discourse representation structure. The resultant logical theory can then be checked for consistency and informativity, and may also be used for question-answering by the third-party reasoning services: the theorem prover Otter and the model builders Mace4 and Satchmo.

The PENG system is based on a client-server architecture and consists of four main components: the text editor, a controlled language processor, a server and the reasoning services (Schwitter 2004b). The text editor communicates with the controlled language processor *via* a socket interface. The processor is running as a client and is connected *via* the server with the reasoning services that are running separate client processes. The server implements a blackboard upon which the processor writes a specification text or 'theory'. The theorem prover searches for a proof for the theory and the model builder looks for a counter-model.

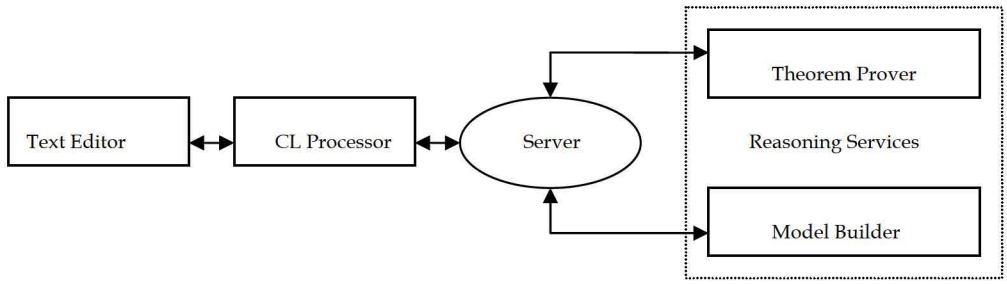


Figure 1: PENG architecture

As part of the Intelligence Analysis Discipline, the approach we intend to take – as described by our information processing architecture – involves extracting information from natural language documents and then formally representing this information using a knowledge representation language. One disadvantage is the loss of (possibly valuable) information brought about by the extraction process. Another is that the formal languages used by the Knowledge Representation and Reasoning community are often difficult for non-specialists to learn and use. In (Schwitter 2004b) it is claimed that these disadvantages can be overcome by machine-processable controlled natural languages which combine the advantages of both formal and natural languages. We believe that this claim is worth investigating. Although PENG is very much a prototype, it still shows potential. If a number of issues can be overcome, we believe the system could make a useful tool.

This report is structured as follows: in Section 2 we describe the controlled language and the PENG text editor, focussing on the editor's paraphrase and look-ahead features. In Section 3 we look at the language processing aspects of PENG. We describe its grammar, its chart parsing technique, its method of discourse representation and the subsequent translation of this representation into first-order logic. In order to make the document self-contained, we have included a large amount of background material in Section 3. We examine a number of grammar formalisms, describe various chart parsing strategies and provide a brief introduction to discourse representation theory and first-order logic. In Section 4 we look at PENG's reasoning services and discuss the particular proof methods these services implement. In Section 5 we draw conclusions.

We make two short remarks before we begin. Firstly, the PENG system is available for use online, requiring a login and password (Schwitter 2007c). Secondly, the author has taken some liberties regarding the 'Dreadsbury Mansion Mystery', a logic puzzle designed to test automated theorem provers (Pelletier 1986). PENG's approach to solving the original puzzle is described in (Schwitter and Ljungberg 2002; Schwitter, Ljungberg et al. 2003).

1.1 Glossary

Anaphora	An instance of one expression referring back to another.
Assignment Function	A function which maps variables to a given model domain. An assignment function can be thought to assign context.
Context-Free Grammar	A generative grammar such that the left-hand side of each rule consists of a single non-terminal symbol.
Chart Parser	A parser which avoids any repetition of work by storing information about previously analysed substrings in the form of a chart.
Definite Clause Grammar (DCG) Notation	A mechanism for implementing context free and unification-based grammars in the logic programming language Prolog.
Discourse Representation Structure (DRS)	A framework for representing contextual dependence within discourse.
Feature Structure	A set of attribute-value pairs where the values themselves may be feature structures.
Formula	A type of description which is built in a well-defined way using: elements from a vocabulary; a (possibly infinite) set of variables; various connectives, punctuation marks and other symbols; and sometimes quantifiers.
Gap Threading	A technique used by context free and unification-based grammars which rules out grammatically incorrect relative clauses.
Generative Grammar	A set of rules which describe how the strings of a particular language can be generated.
Grammar Recogniser	A program which checks whether a given string is grammatically correct but – unlike a parser – does not provide any information regarding the string's syntactic structure.
Interpretation	An interpretation of a vocabulary element is the semantic value in the model domain assigned to it by the interpretation function. An interpretation of a variable is the value in the model domain assigned to it by the assignment function.
Interpretation Function	A model's interpretation function maps each symbol in a given vocabulary to a semantic value in the model domain.
Model	A situation defined by a pair specifying a non-empty domain and interpretation function. There can be multiple models for a given vocabulary with differing domains and interpretation functions.
Model Domain	Any set of real or imaginary things which are of interest, *e.g.* individuals, places, or objects.
Model Builder	A program which accepts a formula as input and attempts to build a finite model that satisfies it.
Parser	A program which analyses the syntactic structure of a given string in order to determine whether the string is grammatically correct.

Satisfiability	Given a model of a particular vocabulary and an assignment function which maps variables to elements of the model domain, a formula (over the same vocabulary) is said to be satisfied in the model if a formula-specific configuration of the interpreted formula elements corresponds with the model itself.
Theorem Prover	A program which determines whether a given formula is valid.
Unification	An operation which takes two feature structures as input and returns – if the structures are compatible – a merged structure.
Unification-Based Grammar	A context-free grammar augmented with feature structures.
Valid Formula	Given the set of all possible models for a particular vocabulary, a formula (over that same vocabulary) is said to be valid if it is satisfied in every model of the set given any assignment function.
Vocabulary	A set of predicate, function and constant symbols.

2. Text Editing

Before we look at the features of the text editor, we take a brief look at the lexicon of PENG.

2.1 The PENG Lexicon

The language restrictions are defined by a unification-based grammar and lexicon. The unification-based grammar defines the structure of simple PENG sentences and states how more complex sentences can be formed using coordinating and subordinating conjunctions. Informally, the set of PENG sentences is restricted to a subset of first-order-logic-representable English natural language sentences. We make some comments about the impact this restriction has on sentence construction later in Section 3.7. We also provide example PENG sentences in Appendix A.

The controlled 'base' lexicon consists of the following (Schwitter, Ljungberg et al. 2003; Schwitter and Tilbrook 2006).

1. Predefined function words which are: determiners *a*, *all*, *an*, *every*, *no* and *the*; the negation function word *not*; the cardinals *one*, *two*, *three*, *four*, *five*, *six*, *seven*, *eight*, *nine*, *ten*; the words *each* and *together* used for disambiguation; the auxiliary words *do* and *does*; the connectives *and*, *if*, *or* and *then*; the prepositions *about*, *around*, *at*, *by*, *for*, *in*, *like*, *of*, *on*, *over*, *than*, *to* and *with*; the copulas *are* and *is*; the query words *how*, *what*, *when*, *where* and *who*; and the relative pronouns *that*, *which* and *who*.
2. Approximately 3,000 of the most commonly used words in English. These predefined content words include nouns, proper nouns, verbs, adjectives and adverbs.

Except for any illegal word, user-defined content words can be added to the base lexicon during the writing process. Illegal words – according to (Schwitter 2007b) – are the nouns *belief* and *wish*; the verbs *can*, *could*, *should*, *might*, *must*, *have to*, *ought to*, *believe*, *want* and *wish*;

the adjective; *former* and the adverb *possibly*. All personal pronouns (*e.g. he, she, you, I, we, they* and *them*) are also illegal. Apart from the full-stop at the end of the sentence, and the question-mark at the end of a question, there is no punctuation.

Synonyms may be defined for nouns. Both proper nouns and nouns are classified as either singular or plural and are assigned one of the following types: person, time or entity. For example *Agatha, Americans* and *politicians* are all of type person, whereas *Newcastle, building* and *car* are of type entity. *Friday, day* and *time* are all of type time. Proper nouns of type person are assigned a gender: neuter, masculine or feminine. Singular nouns of type entity are assigned either an atomic or mass structure. For example *town, word, knife* and *argument* are assigned an atomic structure, whereas *blood, energy, knowledge* and *money* are assigned a mass structure.

Verbs in PENG are classified as intransitive verbs, transitive verbs, prepositional transitive verbs, ditransitive verbs, or prepositional ditransitive verbs. An intransitive verb is a verb which has no direct object (*e.g. dances, falls, votes, dreams, thinks* and *stands*). A transitive verb requires a direct object in the form of a noun phrase (*e.g. drives, has, likes, arranges, questions* and *wants*). A prepositional transitive verb is a transitive verb with an assigned preposition that requires a direct object (*e.g. adapts, appears in, goes to, asks for, behaves like* and *cares for*). A ditransitive verb requires both a direct object and an indirect object (*e.g. tells, offers, regards, combines, sends* and *sticks*). A prepositional ditransitive verb is a ditransitive verb with an assigned preposition that requires both a direct object and an indirect object (*e.g. gives to, hands over, offers to, sends for, combines with* and *tells to*).

All verb forms in PENG are classified as either singular or plural and either finite or infinite. Moreover, all verb forms are classified as either having an event or state structure. Event verbs denote a change in time (*e.g. achieves, begins, goes, walks, sings, travels* and *visits*), whereas state verbs express static properties (*e.g. has, differs, exists, lives, stands, consists* and *waits*).

Adjectives in PENG are single word adjectives (*e.g. good, obvious, yellow, correct, temporary* and *difficult*) or prepositional adjectives. Prepositional adjectives are adjectives which require a preposition (*e.g. afraid of, aware of, happier than, healthier than* and *richer than*). Prepositional adjectives are classified as either comparative or non-comparative.

Adverbs in PENG are each assigned one of the following roles: location (*e.g. anywhere, close, everywhere* and *here*), direction (*e.g. above, backwards, forwards* and *further*), time (*e.g. afterwards, before, eventually* and *forever*), duration (*e.g. longer*), frequency (*e.g. again, daily, enough* and *frequently*), or manner (*e.g. abruptly, angrily, bitterly* and *dramatically*).

2.2 ECOLE

The text editor, called ECOLE – the name derived from 'a look-ahead Editor for a COntrolled LanguagE' – has been designed especially for the PENG language. As discussed in (Schwitter, Ljungberg et al. 2003; Schwitter and Tilbrook 2006), ECOLE's interface consists of three fields: the text, response and query fields. A screen-shot is shown in Figure 2. The text field is where the author writes text in controlled natural language, and is also where look-ahead categories are displayed after each sentence construct is entered. The look-ahead categories are

generated on-the-fly by the controlled language processor. For example, our text field might look like this[1]

```
The fog hangs over Dreadsbury Mansion. The fog is creepy.
[proper_noun, determiner, cardinal, connective:[If]]
```

Hence we may begin a third sentence using one of the following: a proper noun, a determiner, a cardinal number, or the connective *If*. Each look-ahead category is implemented by a hyperlink; the author can click on the link to display more information about the category needed. The author also has the option of selecting an available sentence construct from a drop-down menu. Another feature within the text field is the tab-completion mechanism; the author types the start of a word and completes it by pressing the tab key. The first key stroke retrieves the first available word and successive strokes will iteratively display other available words.

The response field is where system messages are displayed. A paraphrase is given for each sentence which clarifies the interpretation of the input and resolves any synonyms, acronyms, abbreviations and anaphoric references which have been used in the text. Suppose that within the lexicon the noun *fog* has been identified as a synonym of its main form *mist*, then the following is displayed in the response field.

```
Paraphrase:
The <synonym> mist </synonym> hangs over Dreadsbury Mansion.
<anaphora> The <synonym> mist </synonym> </anaphora> is creepy.
```

The paraphrase indicates that the synonym *fog* has been replaced by its main form *mist*, and that the noun phrase *The mist* is an anaphoric expression which has been previously introduced in the text.

The response field also displays: the structure of the last input sentence in the form of a parse tree; the discourse representation structure for the entire text including its representation in first-order logic; and the model generated by the reasoning services, which determines whether the text is satisfiable.

The query field is where the author can pose questions – in controlled natural language – about the text. Look-ahead categories are also generated within this field to guide the writing process. For example,

```
Is the fog creepy [adverb, connective:[and, or], preposition:[in, on,
with], question_mark:[?]]
```

Once the question is completed, it is translated into first-order logic *via* a discourse representation structure and then answered over the generated model. The text editor features both a spellchecker and an integrated lexical editor which can be used to add new content

[1] Here we have user-defined the word *mist* as a singular noun of entity type and mass structure. We have defined the synonym *fog* for the noun *mist*. We have user-defined the verb *hangs over* as a finite, singular prepositional transitive verb of event structure. We have also user-defined the adjective *creepy*.

words. Only minimal linguistic knowledge is required by the author to add a new word to the lexicon. User-defined words can be deleted from the lexicon, but the user cannot delete words from PENG's base lexicon which consists of the predefined function and 3,000-odd content words.

Figure 2: Screen-shot

3. Language Processing

The controlled language processor of the PENG system implements a unification-based grammar and chart parser. Section 3.5 describes aspects of PENG's grammar and its incremental chart parsing techniques; it also discusses how the parser generates look-ahead categories on-the-fly. Sections 3.1-3.4 build the background needed for these sections. Section 3.1 gives a brief introduction to grammar formalisms, focussing on context-free grammars. Section 3.2 introduces chart parsing. Section 3.3 describes unification-based grammars – essentially constrained context-free grammars – and shows how chart parsing strategies for these grammars can be adapted. Section 3.4 discusses some basics of the logic programming language Prolog, and then looks at the definite clause grammar notation which allows us to implement unification-based grammars within Prolog.

Apart from checking PENG text for grammatical correctness and generating look-ahead categories, the PENG chart parser also translates text into Discourse Representation Structures (DRSs). These structures capture the semantic content of the original text and are later translated into first-order logic. Formulae of this logic can then be checked for consistency and questioned-answered by the reasoning services. In Section 3.6 we give a brief introduction to discourse representation theory and describe Hans Kamp and Uwe Reyle's original DRS construction algorithm. We show how a DRS can be translated into first-order logic and describe how PENG's chart parser constructs a flattened DRS using a variation of the original DRS construction algorithm.

In Section 3.7 we examine 'nonfirstorderisable' English sentences. Such sentences cannot be represented in first-order logic, hence they cannot be formulated in the PENG language. We discuss the impact this has on text composition.

3.1 Context-Free Grammars

A grammar provides a precise description of a formal language represented by a set of strings. As discussed in (Gilbert 1966) there are two main types of grammars: generative and analytic. We'll describe analytic grammars shortly.

Generative grammars – also known as phrase structure grammars – are sets of rules which tell us how strings of a particular language can be generated. In the classic formalisation first proposed in (Chomsky 1956), a generative grammar G is defined as a 4-tuple (N, Σ, P, S) where N is a finite set of non-terminal symbols, Σ is a finite set of terminal symbols that is disjoint from N, P is a finite set of production rules where each rule is of the form $(\Sigma \cup N)^* N (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$, and S is a distinguished start symbol belonging to N. Here $*$ is the Kleene star operator[2] and $\cup$ denotes set union. The language of G – written $L(G)$ – is defined as the set of strings over Σ which are formed by starting with S and applying the production rules until no non-terminal symbols remain. An example is the grammar G with $N \equiv \{S, B\}$, $\Sigma \equiv \{a,b,c\}$, starting symbol S and the following production rules.

[2] The star operator works as follows: if V is a set of symbols, then V^* is the set of all strings over symbols in V, including the empty string.

1. $S \rightarrow aBSc$
2. $S \rightarrow abc$
3. $Ba \rightarrow aB$
4. $Bb \rightarrow bb$

Two possible strings generated by S are *abc* and *aabbcc*. The first string can be generated directly by application of rule 2. The second string can be generated *via* $S \Rightarrow_1 aBSc \Rightarrow_2 aBabcc \Rightarrow_3 aaBbcc \Rightarrow_4 aabbcc$. Here $X \Rightarrow_i Y$ has the meaning that X generates Y by application of rule i. We can see that the grammar defines the language $L(G) \equiv \{a^n b^n c^n \mid n \geq 1\}$.

A generative grammar is said to be incomplete if we can derive a string from S containing one or more non-terminal symbols, but can find no production rule to apply to the string. Moreover, the grammar is said to be recursive if it contains a non-terminal symbol which can be recursively derived from a production rule.

A context-free grammar is a generative grammar such that the left-hand side of the production rule consists of a single non-terminal symbol. As an example, consider the grammar with the set of non-terminal symbols $N \equiv \{S, np, det, n, vp, iv, pp\}$, the set of terminal words and phrases $\Sigma \equiv \{the, woman, lives, in\ Dreadsbury\ Mansion\}$, starting symbol S and the following production rules.

1. $S \rightarrow np\ vp$
2. $np \rightarrow det\ n$
3. $det \rightarrow the$
4. $n \rightarrow woman$
5. $vp \rightarrow iv\ pp$
6. $iv \rightarrow lives$
7. $pp \rightarrow in\ Dreadsbury\ Mansion$

Here the symbols S, np, vp, det, n, iv and pp abbreviate the grammatical categories 'sentence', 'noun phrase', 'verb phrase', 'determiner', 'noun', 'intransitive verb' and 'prepositional phrase' respectively. We can see from the production rules that S generates the string *the woman lives in Dreadsbury Mansion*. We can represent the syntactic structure of the string – as described by the grammar – in the form of a tree. Here S is the root node, the elements of N are the branch nodes and the elements of Σ are the leaf nodes.

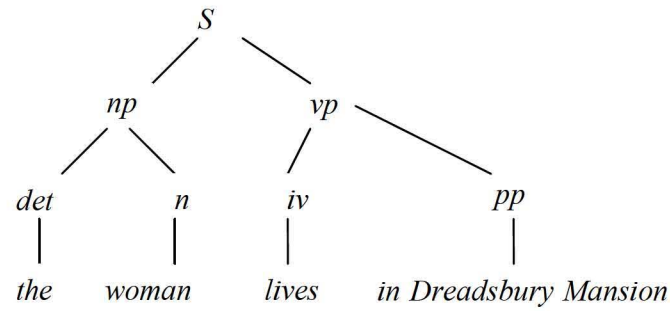


Figure 3: Parse tree

A context-free parser is used to construct such a tree. Given a context-free grammar and an input string belonging to the language of the grammar, a parsing algorithm builds a parse tree top-down such that *S* is the root node and every node of the tree is allowable by the grammar. If a tree can be constructed such that the string is listed in the correct order at the leaf nodes, then the parser deems the string to be well-formed or grammatically correct. In other words, a parser analyses the syntactic structure of a given string in order to determine whether the string is grammatically correct.

Most parsing algorithms assume the language to be parsed is described by means of a generative grammar. There is usually no correspondence between the algorithm used to parse the language and the generative grammar itself. In contrast, analytic grammars are sets of rules which tell us how strings can be analysed in order to determine whether they are members of a particular language. Essentially, these grammars formally describe a parser for a language; they describe how a language is to be read, rather than how it is to be written. Since analytic grammars have no further bearing on this report, in the sequel any reference to 'grammar' should be interpreted as 'generative grammar'.

A major drawback of naïve parsing algorithms is that they often build and discard the same sub-tree multiple times. For example, suppose we have a context-free grammar with $N \equiv \{S, np, det, n, rc, vp, iv, pp\}$, $\Sigma \equiv \{the, woman, that, lives, in\ Dreadsbury\ Mansion\}$, starting symbol *S* and the following production rules.

1. $S \rightarrow np\ vp$
2. $np \rightarrow det\ n\ rc$
3. $np \rightarrow det\ n$
4. $rc \rightarrow relpro\ vp$
5. $vp \rightarrow iv\ pp$
6. $det \rightarrow the$
7. $n \rightarrow woman$
8. $relpro \rightarrow that$
9. $iv \rightarrow lives$
10. $pp \rightarrow in\ Dreadsbury\ Mansion$

Note that *rc* and *relpro* abbreviate the grammatical categories 'relative clause' and 'relative pronoun' respectively. Given the input string *the woman lives in Dreadsbury Mansion*, a naïve parser will first attempt to apply the rule $np \rightarrow det\ n\ rc$ (since it is the first applicable production rule in the list). Failing this, it will attempt to apply $np \rightarrow det\ n$. This means the parser repeats the analysis of both determiner and noun at each rule application. A work-around this redundancy is chart parsing. Since PENG implements an incremental chart parser – incremental in that it allows modifications to be made to the input string on-the-fly – we now give a brief introduction to the chart parsing method. Much of the next section follows from (Gazdar and Mellish 1990; Blackburn and Striegnitz 2002).

3.2 Chart Parsing Basics

A chart parser is a parser which implements a chart. Essentially, a chart stores information about substrings the parser has already analysed. The chart parsing algorithm checks the chart to see whether it has already produced an analysis of any substring it is parsing; if it has, the algorithm uses this information and hence avoids any repetition of work.

Suppose we read from left to right the string of terminal symbols – or in our case, words – found at the leaf nodes of a context-free parse tree. This string can be represented by a Well-Formed Substring Table (WFST). If we assign the beginning and end of the string the indices 0 and n, and assign the spaces between string constituents the indices numbered 1 to $n - 1$ from left to right, then a WFST tells us for each pair of indices (i, j) – where $0 \leq I < j \leq n$ – what set of non-terminal symbols span the substring of words found between i and j. As an example, consider the context-free grammar with $N \equiv \{S, np, det, n, vp, iv, adv\}$, $\Sigma \equiv \{the, butler, acts, suspiciously\}$, starting symbol S and the following production rules.

1. $S \rightarrow np\ vp$
2. $np \rightarrow det\ n$
3. $det \rightarrow the$
4. $n \rightarrow butler$
5. $vp \rightarrow iv\ adv$
6. $iv \rightarrow acts$
7. $adv \rightarrow suspiciously$

The string *the butler acts suspiciously* generated by S can be represented by the following WFST.

$i \backslash j$	1	2	3	4
0	*det*	*np*		*S*
1		*n*		
2			*iv*	*vp*
3				*adv*

Figure 4: WFST

We can think of a WFST as a graph whereby arcs between nodes i and j – where $0 \leq i < j \leq n$ – are labelled with the non-terminal symbols that span the substring of words between nodes i and j. Below we give a graph representation for our current example.

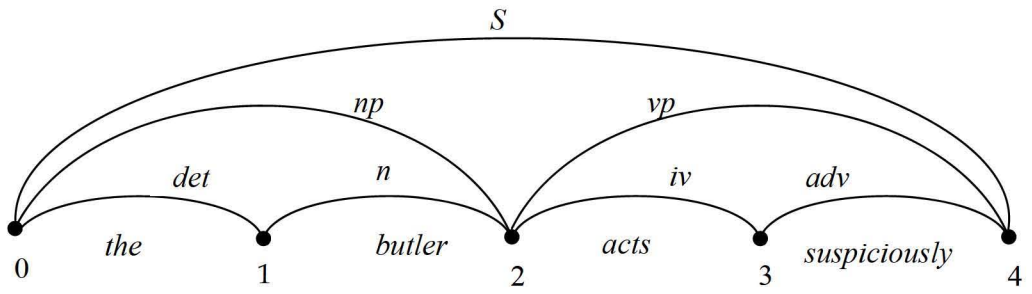


Figure 5: Graph representation

Given a context-free grammar and an input string, a chart parser parses the string by constructing a variation of a WFST called a chart. Note that for readability, we will describe a chart using a graph representation; ordinarily however, a chart is implemented as a look-up table. In a chart, arcs are usually referred to as edges. Edges are directed clockwise and are labelled with dotted rules (these are explained shortly). Edges may also be empty, *i.e.* they loop back on themselves. The chart parsing process begins with the parser constructing an initial chart. The initial chart varies according to the type of parser. The parser then employs a set of rules to heuristically decide when an edge should be added to the chart. This set of rules – along with the specification of when they should be applied – forms a strategy. When the parser finds an edge which spans the entire string, it has succeeded in parsing the string. There may be further parses to find, but the parser has found at least one.

The dotted rule notation was introduced in (Earley 1970). Essentially an edge labelled by a dotted rule represents a hypothesised string constituent. The string constituent can be fully analysed, partially analysed, or completely analysed by the chart parser. Suppose $S \rightarrow np\ vp$ and $det \rightarrow the$ are production rules of the grammar, then the following dotted rules can be used as edge labels.

$S \rightarrow \bullet\ np\ vp$

$S \rightarrow np \bullet vp$

$S \rightarrow np\ vp \bullet$

$det \rightarrow the \bullet$

The dot within these labels indicates the extent to which the hypothesis that the rule is applicable has been verified by the chart parser. Rules of the form $S \rightarrow \bullet\ np\ vp$ are only used to label empty edges. This particular rule denotes the hypothesis that S can be found spanning a substring that represents a *np vp* sequence. The rule $S \rightarrow np \bullet vp$ denotes a similar hypothesis, however in this case the hypothesis has been partially confirmed; the *np* sequence has already been analysed by the parser. The dotted rules $S \rightarrow np\ vp \bullet$ and $det \rightarrow the \bullet$ denote fully confirmed hypotheses, whereby both the *np vp* sequence and terminal word *the* have been analysed. Note that a rule such as $det \rightarrow \bullet\ the$ semantically makes sense – it denotes the hypothesis that the determiner can be found spanning a substring that represents the terminal word *the* – however we ignore rules of this form here, since they are not necessary for chart parsing.

Our WFST above can be represented as the following chart.

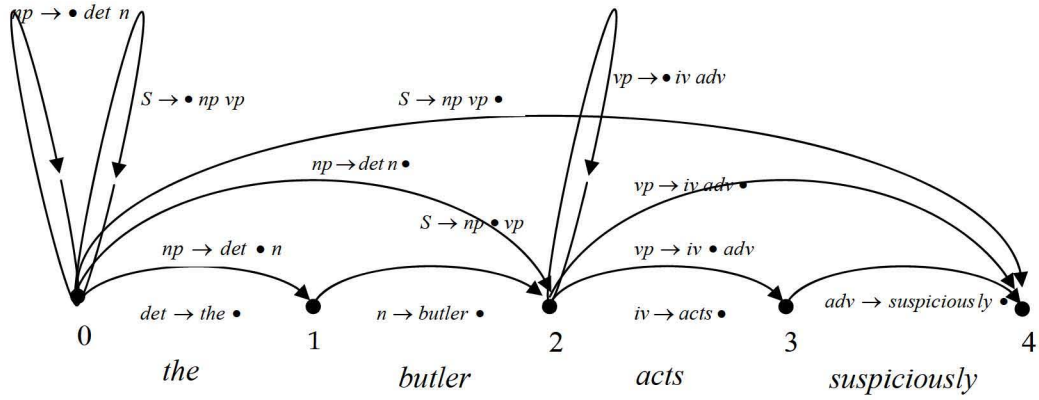


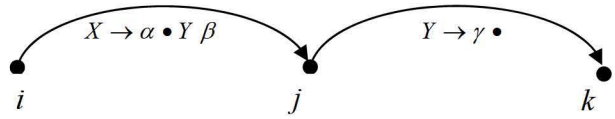
Figure 6: Completed chart

Edges of a chart that represent unconfirmed hypotheses are called active, whereas those that represent confirmed hypotheses are inactive. For example in the chart above, the edges labelled $np \rightarrow det \bullet n$ and $S \rightarrow \bullet\ np\ vp$ are active, whereas the edges labelled $vp \rightarrow iv\ adv \bullet$ and $det \rightarrow the \bullet$ are inactive.

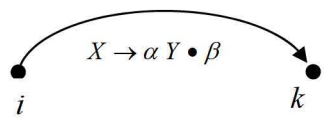
Edges can be described using the notation (i, j, L) where i is the start position of the edge, j is the end position and L is the label of the edge. For example the edge labelled $np \rightarrow det \bullet n$ in Figure 6 can be written $(0, 1, np \rightarrow det \bullet n)$, whereas the edge labelled $vp \rightarrow iv\ adv \bullet$ can be written $(2, 4, vp \rightarrow iv\ adv \bullet)$.

3.2.1 The Fundamental Rule

Every chart parsing strategy incorporates one rule in particular. This rule is called the fundamental rule of chart parsing. This rule combines: (1) an active edge whose label hypothesises a non-terminal symbol Y, with (2) an inactive edge whose label features Y on the left-hand side. The result is a new edge which spans both the original active and inactive edges. Formally, suppose the chart contains the following edges where $0 \leq i < j \leq n$, X and Y are non-terminal symbols, α is a (possibly empty) sequence of non-terminal symbols, β is a sequence of non-terminal symbols, and γ is either a terminal symbol or a sequence of non-terminal symbols.



Then the parser adds the following edge.



We can see from Figure 6 that edges (0, 0, $S \rightarrow \bullet\ np\ vp$) and (0, 2, $np \rightarrow det\ n\ \bullet$) combine to form (0, 2, $S \rightarrow np \bullet vp$), edges (0, 1, $np \rightarrow det \bullet n$) and (1, 2, $n \rightarrow butler\ \bullet$) combine to form (0, 2, $np \rightarrow det\ n\ \bullet$), and edges (0, 2, $S \rightarrow np \bullet vp$) and (2, 4, $vp \rightarrow iv\ adv\ \bullet$) combine to form (0, 4, $S \rightarrow np\ vp\ \bullet$).

3.2.2 A General Algorithm

Chart parsing is either conducted top-down or bottom-up. In top-down chart parsing the parser start with *S* and tries to transform it into the input string. The parser takes grammatical categories and breaks them into smaller constituents and eventual terminal symbols. In bottom-up chart parsing, the parser starts with the input string and tries to rewrite it to *S*. It takes each terminal symbol and attempts to locate the parent grammatical categories the symbol belongs to. Once found, the parser then attempts to locate *those* categories' parents, and so on.

Chart parsing often relies on a data-structure called an agenda. The parser stores the remaining (not yet analysed) edges in the agenda. It then adds these edges one at a time to the chart, using them to build new edges *via* the fundamental rule. Here we will treat the agenda as a stack; we push edges on the agenda and pop edges from the agenda in a last-in, first-out manner. Alternatively, we could treat the agenda as a queue whereby edges are taken in a first-in, first-out manner. Treating the agenda as a stack results in a depth-first search strategy, whereas treating the agenda as a queue results in a breadth-first search strategy. Hence the order of the edges in the agenda is of vital importance.

A general algorithm for both top-down and bottom-up chart parsing can be described as follows.

1. Construct the initial agenda and chart.
2. Repeat steps a, b and c until the agenda is empty.
    a. Pop the first edge from the agenda and – as long as it is not already there – add it to the chart. This edge becomes the current edge.
    b. If possible, apply the fundamental rule in order to combine the current edge with any other edges from the chart. New edges formed should be pushed on the agenda.
    c. Make hypotheses – in the form of active edges – about new sentence constituents based on the current edge and the rules of the grammar. Push these new edges on the agenda.
3. If the chart contains an inactive edge from the first node to the last with label $S \rightarrow \gamma\ \bullet$ – where γ is either a terminal symbol or a (possible sequence of) non-terminal symbol(s) – then we have successfully parsed the sentence, else we have failed.

How Steps 1 and 2c are carried out distinguishes between top-down and bottom-up chart parsing strategies. We'll shortly look at an example to explain the difference between the two strategies.

Neither the top-down nor bottom-up parsing strategy is considered better than the other. As mentioned in (Longley and Stark 2002), top-down parsers are a little less powerful but their

algorithms are easier to implement. Although PENG features a basic (top-down) parser which implements the general algorithm described previously, it is worth mentioning that more sophisticated chart parsing algorithms exist. Two popular algorithms are the Cocke-Younger-Kasami (CYK) algorithm and the Earley algorithm. CYK is a bottom-up algorithm and requires its context-free grammar to be written in Chomsky Normal Form (Kasami 1965; Younger 1967). A definition of Chomsky Normal Form can be found at (Autebert, Berstel et al. 1997). The Earley algorithm is a top-down algorithm which is somewhat faster and more efficient than CYK (Earley 1970).

3.2.3 The Top-Down Strategy

Since PENG implements a top-down chart parser, we will build a chart in all its gory detail following the general algorithm using the top-down strategy. We avoid doing this for the bottom-up case, however we will comment briefly on how the general algorithm can be adapted for this strategy. Our example grammar will again be used in Section 3.5.2 where we describe how PENG handles modification (*i.e.* insertion, deletion and replacement) of the input string on-the-fly. Suppose we have the following context-free grammar with the set of non-terminal symbols $N \equiv \{S, np, det, n, rc, relpro, vp, iv, adv\}$, the set of terminal words $\Sigma \equiv \{the, butler, that, acts, suspiciously\}$, starting symbol S and the following production rules.

1. $S \rightarrow np\ vp$
2. $np \rightarrow det\ n\ rc$
3. $np \rightarrow det\ n$
4. $rc \rightarrow relpro\ vp$
5. $vp \rightarrow iv\ adv$
6. $det \rightarrow the$
7. $n \rightarrow butler$
8. $relpro \rightarrow that$
9. $iv \rightarrow acts$
10. $adv \rightarrow suspiciously$

For now, suppose we have the input string *the butler acts suspiciously*.

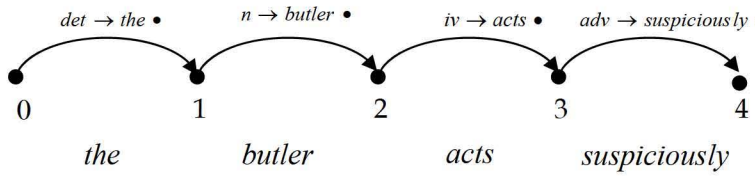
When constructing an initial agenda for top-down chart parsing, we select the grammar production rules that feature S. We form an active empty edge for each rule – at node 0 – and place each edge in the empty agenda/stack. Following our example, we have the initial agenda

$$(0, 0, S \rightarrow \bullet\ np\ vp)$$

Note that if we had a grammar production rule such as $S \rightarrow S\ coord\ S$ where *coord* is the word *and* or *or*, say, then we would also add a similar active empty edge $(0, 0, S \rightarrow \bullet\ S\ coord\ S)$ to the agenda.

The initial chart is comprised of the inactive edges featuring the terminal words of the input string. Essentially the top-down strategy involves breaking down grammatical categories into constituent categories by hypothesising new edges from active edges. Hence we work our

way down from active edges featuring grammatical categories to these smaller inactive edges featuring the terminal words. Following our example, we have the initial chart



Step 2c of the general chart parsing algorithm is adapted to the top-down strategy in the following way. If you are adding the current active edge $(i, j, X \rightarrow \alpha \bullet Y \beta)$ to the chart, then for every grammar rule $Y \rightarrow \gamma$, push the edge $(i, j, Y \rightarrow \bullet \gamma)$ on the agenda. Here i and j are node labels, X and Y are non-terminal symbols, α is a (possibly empty) sequence of non-terminal symbols, and β and γ are sequences of non-terminal symbols.

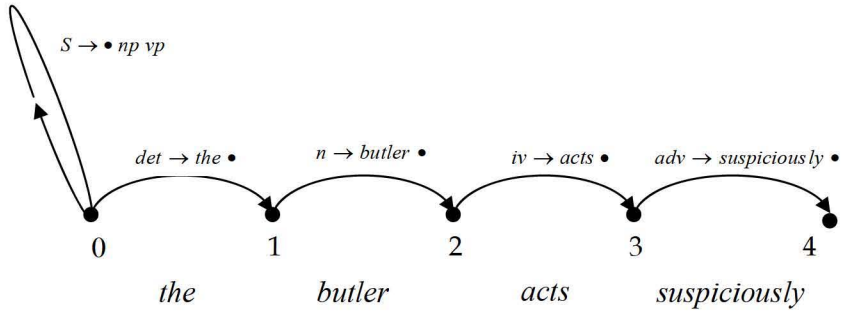
Round 1

2a) We pop edge (0, 0, $S \rightarrow \bullet$ *np vp*) from our initial agenda and add it to the chart.

2b) We are unable to apply the fundamental rule, hence we move to Step 2c.

2c) Our current edge is (0, 0, $S \rightarrow \bullet$ *np vp*) and we have the two grammar rules $np \rightarrow$ *det n rc* and $np \rightarrow$ *det n*. Following Step 2c outlined above, we push the edges (0, 0, $np \rightarrow \bullet$ *det n rc*) and (0, 0, $np \rightarrow \bullet$ *det n*) on the agenda.

After Round 1 we have the chart



We have the following agenda.

(0, 0, $np \rightarrow \bullet$ *det n*)
(0, 0, $np \rightarrow \bullet$ *det n rc*)

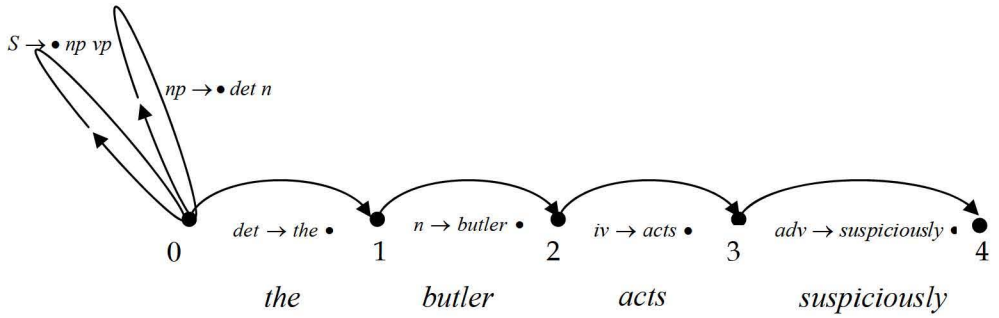
Round 2

2a) We pop the edge (0, 0, $np \rightarrow \bullet$ *det n*) from the agenda and add it to the chart.

2b) We can apply the fundamental rule by combining (0, 0, $np \rightarrow \bullet$ *det n*) and (0, 1, *det* → *the* •) to form the new edge (0, 1, $np \rightarrow$ *det* • *n*) which we push on the agenda.

2c) Our current edge is (0, 0, $np \rightarrow \bullet$ *det n*), but there are no grammar rules with *det* on the left-hand side and non-terminals on the right, so nothing is pushed on the agenda at this step.

After Round 2 we have the chart



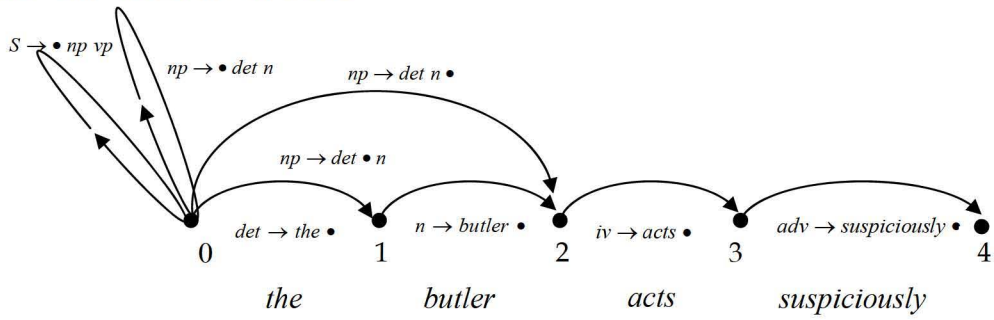
We have the following agenda.

(0, 1, *np* → *det* • *n*)
(0, 0, *np* → • *det n rc*)

Round 3
2a) We pop (0, 1, *np* → *det* • *n*) from the agenda and add it to the chart.
2b) We can apply the fundamental rule by combining (0, 1, *np* → *det* • *n*) and (1, 2, *n* → *butler* •) to form the new edge (0, 2, *np* → *det n* •) which we push on the agenda.
2c) Our current edge is (0, 1, *np* → *det* • *n*), but there are no grammar rules with *n* on the left-hand side and non-terminals on the right. Hence we move to Round 4.

Round 4
2a) We add (0, 2, *np* → *det n* •)to the chart.
2b) We combine (0, 0, *S* → • *np vp*) and (0, 2, *np* → *det n* •) to form the new edge (0, 2, *S* → *np* • *vp*) which we push on the agenda.
2c) No new hypothesis can be made from our current inactive edge (0, 2, *np* → *det n* •).

After Round 4 we have the chart



We have the following agenda.

(0, 2, *S* → *np* • *vp*)
(0, 0, *np* → • *det n rc*)

Round 5

2a) We add (0, 2, $S \rightarrow np \bullet vp$) to the chart.

2b) We are unable to apply the fundamental rule, hence we move to Step 2c.

2c) Our current edge is (0, 2, $S \rightarrow np \bullet vp$) and we have the grammar rule $vp \rightarrow iv\ adv$. Following Step 2c, we push the edge (2, 2, $vp \rightarrow \bullet\ iv\ adv$) on the agenda.

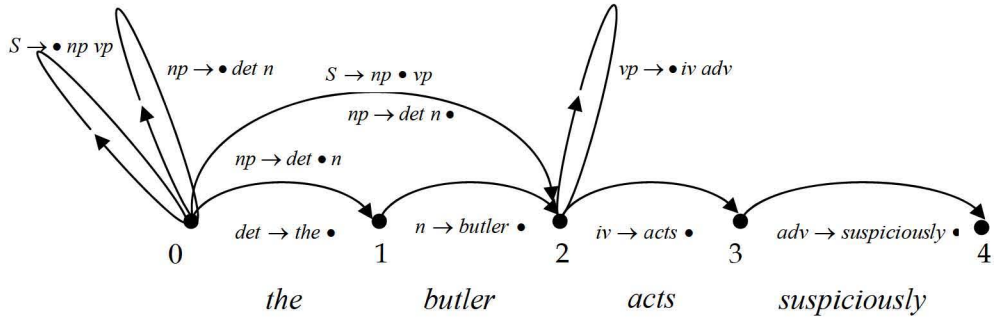
Round 6

2a) We add (2, 2, $vp \rightarrow \bullet\ iv\ adv$) to the chart.

2b) We combine (2, 2, $vp \rightarrow \bullet\ iv\ adv$) and (2, 3, $iv \rightarrow acts \bullet$) to form the new edge (2, 3, $vp \rightarrow iv \bullet adv$) which we push on the agenda.

2c) There are no grammar rules with *iv* on the left-hand side and non-terminals on the right.

After Round 6 we have the chart



We have the following agenda.

(2, 3, $vp \rightarrow iv \bullet adv$)
(0, 0, $np \rightarrow \bullet\ det\ n\ rc$)

Round 7

2a) We add (2, 3, $vp \rightarrow iv \bullet adv$) to the chart.

2b) We combine (2, 3, $vp \rightarrow iv \bullet adv$) and (3, 4, $adv \rightarrow suspiciously \bullet$) to form the new edge (2, 4, $vp \rightarrow iv\ adv \bullet$) which we push on the agenda.

2c) There are no grammar rules with *adv* on the left-hand side and non-terminals on the right.

Hence we move to Round 8.

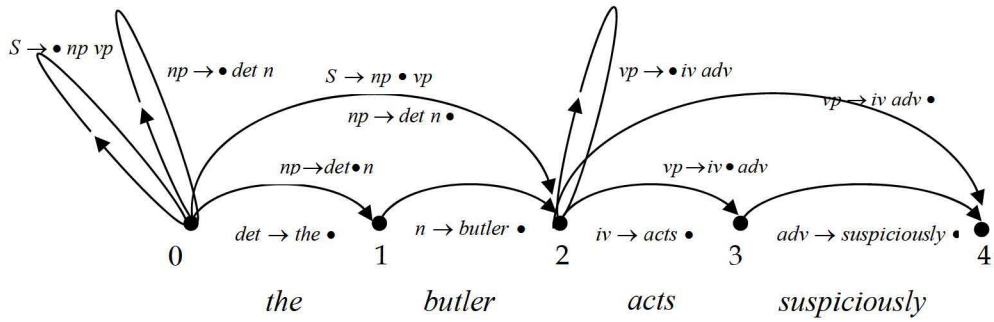
Round 8

2a) We add (2, 4, $vp \rightarrow iv\ adv \bullet$) to the chart.

2b) We combine (0, 2, $S \rightarrow np \bullet vp$) and (2, 4, $vp \rightarrow iv\ adv \bullet$) to form the new edge (0, 4, $S \rightarrow np\ vp \bullet$) which we push on the agenda.

2c) No new hypothesis can be made from our current inactive edge (2, 4, $vp \rightarrow iv\ adv \bullet$).

After Round 8 we have the chart



We have the following agenda.

(0, 4, $S \rightarrow np\ vp$ •)
(0, 0, $np \rightarrow$ • $det\ n\ rc$)

Round 9

2a) We add (0, 4, $S \rightarrow np\ vp$ •) to the chart.

2b) We are unable to apply the fundamental rule, hence we move to Step 2c.

2c) No new hypothesis can be made from our current inactive edge (0, 4, $S \rightarrow np\ vp$ •).

Round 10

2a) We add (0, 0, $np \rightarrow$ • $det\ n\ rc$) to the chart.

2b) We combine (0, 0, $np \rightarrow$ • $det\ n\ rc$) and (0, 1, $det \rightarrow the$ •) to form the new edge (0, 1, $np \rightarrow det$ • $n\ rc$) which we push on the agenda.

2c) There are no grammar rules with *det* on the left-hand side and non-terminals on the right.

Hence we move to Round 11.

Round 11

2a) We add (0, 1, $np \rightarrow det$ • $n\ rc$) to the chart.

2b) We combine (0, 1, $np \rightarrow det$ • $n\ rc$) and (1, 2, $n \rightarrow butler$ •) to form the new edge (0, 2, $np \rightarrow det\ n$ • rc) which we push on the agenda.

2c) There are no grammar rules with *n* on the left-hand side and non-terminals on the right.

Hence we move to Round 12.

Round 12

2a) We add (0, 2, $np \rightarrow det\ n$ • rc) to the chart.

2b) We are unable to apply the fundamental rule, hence we move to Step 2c.

2c) Our current edge is (0, 2, $np \rightarrow det\ n$ • rc) and we have the grammar rule $rc \rightarrow relpro\ vp$.

Following Step 2c, we push the edge (2, 2, $rc \rightarrow$ • $relpro\ vp$) on the agenda.

Round 13

2a) We add (2, 2, *rc* → • *relpro vp*) to the chart.

2b) We are unable to apply the fundamental rule, hence we move to Step 2c.

2c) There are no grammar rules with *relpro* on the left-hand side and non-terminals on the right. Hence we move to Round 14.

Round 14

The agenda is empty so we move to Step 3.

3) Since the chart contains an inactive edge from the first node to the last – namely (0, 4, *S* → *np vp* •) we return 'success' and are done.

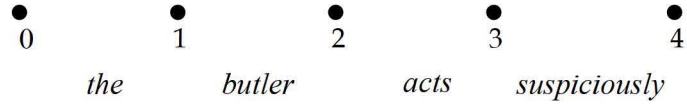
As an aside, it is worth pointing out that if we had the production rules $S \rightarrow np\ vp$ and $S \rightarrow S\ coord\ S$ in our grammar, then at some stage during the parsing process we would get the edge (i, j, $S \rightarrow S\ coord \bullet S$) appearing in the chart for some nodes i and j where $i < j$. This means that *via* Step 2c, we'd eventually introduce the active empty edges (i, j, $S \rightarrow \bullet\ np\ vp$) and ($i$, j, $S \rightarrow \bullet\ S\ coord\ S$) to the chart. Hence this means the parser looks for coordinated (or nested) sentences within the input string, but only according to the production rules of the grammar.

3.2.4 The Bottom-Up Strategy

When constructing an initial agenda for bottom-up chart parsing, we select the grammar production rules that feature the terminal symbols. We form an inactive edge for each rule – which spans the nodes of the terminal symbol – and place each edge in an empty agenda. Following our example, we have the initial agenda

(0, 1, *det* → *the* •)
(1, 2, *n* → *butler* •)
(2, 3, *iv* → *acts* •)
(3, 4, *adv* → *suspiciously* •)

Essentially the bottom-up strategy involves building up the terminal symbols into grammatical categories – and eventually *S* – by hypothesising new edges from inactive edges. The initial chart for bottom-up chart parsing is empty. We can see that this is a reasonable starting point, since the algorithm specifies that edges from the agenda are added one at a time to the chart, and our initial agenda contains inactive edges that feature the terminal symbols. Following our example, we have the initial chart



Step 2c of the general chart parsing algorithm is adapted to the bottom-up strategy in the following way. If you are adding the current inactive edge (i, j, $X \rightarrow \alpha \bullet$) to the chart, then for every grammar rule $Y \rightarrow X\ \beta$, push the edge (i, j, $Y \rightarrow \bullet\ X\ \beta$) on the agenda. Here i and j are

node labels, X and Y are non-terminal symbols, α is either a terminal symbol or a sequence of non-terminal symbols, and β is a sequence of non-terminal symbols.

In order to better describe how the algorithm implements the bottom-up strategy, we will apply the algorithm to our running example for a few rounds.

Round 1

2a) We pop edge (0, 1, *det* → *the* •) from our initial agenda and add it to the chart.

2b) We are unable to apply the fundamental rule, hence we move to Step 2c.

2c) Our current edge is (0, 1, *det* → *the* •) and we have the two grammar rules *np* → *det n rc* and *np* → *det n*. Following Step 2c outlined above, we push the edges (0, 0, *np* → • *det n rc*) and (0, 0, *np* → • *det n*) on the agenda.

Round 2

2a) We pop edge (0, 0, *np* → • *det n*) from our initial agenda and add it to the chart.

2b) We can apply the fundamental rule by combining (0, 0, *np* → • *det n*) and (0, 1, *det* → *the* •) to form the new edge (0, 1, *np* → *det* • *n*) which we push on the agenda.

2c) No new hypothesis can be made from our current active edge (0, 0, *np* → • *det n*).

Round 3

2a) We add edge (0, 1, *np* → *det* • *n*) to the chart.

2b) We are unable to apply the fundamental rule, hence we move to Step 2c.

2c) No new hypothesis can be made from our current active edge (0, 1, *np* → *det* • *n*).

3.3 Unification-Based Grammars

Section 3.1 gave us a general overview of grammars and context-free grammars in particular. This section describes unification-based grammars, which are context-free grammars augmented with constraints called feature structures. After a brief introduction, we show how chart parsing strategies for these grammars can be adapted. In Section 3.4 we look at the definite clause grammar notation which allows us to implement unification-based grammars within the logic programming language Prolog. This section and Section 3.4 provide us with the background needed for an examination of PENG's grammar, its incremental chart parsing techniques and look-ahead category generation. Much of this section is adapted from (Jurafsky and Martin 2000). We begin with an example.

Suppose we have the following context-free grammar with $N \equiv \{S, np, det, n, vp, iv\}$, $\Sigma \equiv \{a, brother, schemes\}$, starting symbol S and the following production rules.

1. $S \rightarrow np\ vp$
2. $np \rightarrow det\ n$
3. $vp \rightarrow iv$
4. $det \rightarrow a$
5. $n \rightarrow brother$
6. $iv \rightarrow schemes$

Suppose we want to extend the grammar in order to generate the string *all brothers scheme*. We could just add *all*, *brothers* and *scheme* to Σ and add the production rules $det \rightarrow all$, $n \rightarrow brothers$ and $iv \rightarrow scheme$, however this new grammar would allow us to generate the unwanted strings, *a brother scheme*, *all brother scheme*, *all brothers schemes* and *a brothers schemes*. Alternatively, we could replace *np*, *n*, *vp* and *iv* with non-terminal symbols representing their plural and singular forms. We'd then have the set of non-terminals $N \equiv \{S, npsg, nppl, det, nsg, npl, vpsg, vppl, ivsg, ivpl\}$, the set of terminals $\Sigma \equiv \{a, all, brother, brothers, schemes, scheme\}$, starting symbol *S*and the following production rules.

1. $S \rightarrow npsg\ vpsg$
2. $S \rightarrow nppl\ vppl$
3. $npsg \rightarrow detsg\ nsg$
4. $nppl \rightarrow detpl\ npl$
5. $vpsg \rightarrow ivsg$
6. $vppl \rightarrow ivpl$
7. $detsg \rightarrow a$
8. $detpl \rightarrow all$
9. $nsg \rightarrow brother$
10. $npl \rightarrow brothers$
11. $ivsg \rightarrow scheme$
12. $ivpl \rightarrow schemes$

A drawback to this method is the drastic increase in the size of the grammar.

A much better solution is to integrate feature structures within the grammar. Here the grammatical category symbols *np*, *vp*, *det*, *n*, *etc.* can be thought of as sets of attributes designating – for example – category type, grammatical number, grammatical person, gender and/or tense. A feature structure is simply a set of attribute-value pairs. Often structures are denoted by an attribute-value matrix. For example, the following matrix captures a restricted subcategory of noun phrases whereby each phrase is singular and in third person.

$$\begin{bmatrix} category & np \\ number & sg \\ person & 3 \end{bmatrix}$$

Values can be atomic – *e.g. np*, *sg* or 3 – or are feature structures themselves. Consider the matrix below.

$$\begin{bmatrix} category & np \\ agree & \begin{bmatrix} number & sg \\ person & 3 \end{bmatrix} \end{bmatrix}$$

Here the 'agreement' attribute *agree* takes a feature structure – consisting of *number* and *person* attribute-value pairs – as its value. Matrices of this form allow us to take two grammatical categories and test their value equality for both *number* and *person* attributes. Moreover in a variation of the above matrix, an *agree* structure may act as a value for a *subject* attribute. This

allows us to take a category and the grammatical subject of category and test their value equality for both *number* and *person* attributes. Note that in general, feature structures acting as values of *agree* can contain attributes other than *number* and *person*. These attributes are typically present because grammatical categories often need to agree (at least) on the values of these attributes.

A feature path is a list of attributes through a feature structure leading to a particular value. Using the last feature structure as an example, the feature path < *agree number* > leads to the value *sg*, whereas < *agree person* > leads to 3. A logical step from the notion of feature paths is the representation of structures as Directed Acyclic Graphs (DAGs). Our feature structure above looks as follows.

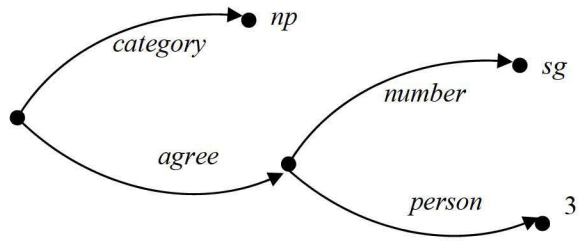


Figure 7: Feature structure represented as directed acyclic graph

Feature structures that share an identical substructure (or node in the DAG) can be represented by re-entrant structures. A symbol ⊗ is used to indicate the shared structure. For example in the following matrix, the *agree* attribute and the < *subject agree* > feature path share the same value, namely the structure consisting of *person* and *number* attribute-value pairs.

$$\begin{bmatrix} agree & \otimes \begin{bmatrix} person & 3 \\ number & sg \end{bmatrix} \\ subject & [agree \quad \otimes] \end{bmatrix}$$

This is equivalent to the following matrix.

$$\begin{bmatrix} agree & \otimes \\ subject & \left[agree \quad \otimes \begin{bmatrix} person & 3 \\ number & sg \end{bmatrix} \right] \end{bmatrix}$$

The symbol ⊗ can be thought of as acting as both label and placeholder. It labels the structure consisting of *person* and *number* attribute-value pairs, and as the value of *agree*, it acts as placeholder for that structure.

Unification is a partial operation on feature structures. The binary operator ∪ takes two feature structures as argument and returns – when successful – a merged structure. If the structures are incompatible, unification fails. We now look at a number of examples – taken from (Jurafsky and Martin 2000) – to illustrate. Since the input structures are identical in the equation below, unification returns the same structure as output.

$$[number \quad sg] \cup [number \quad sg] = [number \quad sg]$$

The next unification fails since the two attributes have incompatible values.

$$[number \quad sg] \cup [number \quad pl] \text{ fails}$$

The [ ] value in the following structure indicates that it has been left unspecified. Such a value is compatible with any value of a corresponding attribute in another structure.

$$[number \quad sg] \cup [number \quad [\,]] = [number \quad sg]$$

The next equation merges two structures; the unification is successful since the structures do not share attributes with incompatible values.

$$[number \quad sg] \cup [person \quad 3] = \begin{bmatrix} number & sg \\ person & 3 \end{bmatrix}$$

In the following equation, the *agree* attribute of the left-hand side structure receives a value as a result of unification.

$$\begin{bmatrix} agree & \otimes \\ subject & [agree \quad \otimes [\,]] \end{bmatrix} \cup \begin{bmatrix} subject & \begin{bmatrix} agree & \begin{bmatrix} person & 3 \\ number & sg \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

$$= \begin{bmatrix} agree & \otimes \\ subject & \begin{bmatrix} agree & \otimes \begin{bmatrix} person & 3 \\ number & sg \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

In the next example, unification fails since the values found *via* < *subject agree number* > are incompatible.

$$\begin{bmatrix} agree & \otimes \begin{bmatrix} number & sg \\ person & 3 \end{bmatrix} \\ subject & [agree \quad \otimes] \end{bmatrix} \cup \begin{bmatrix} agree & \begin{bmatrix} number & sg \\ person & 3 \end{bmatrix} \\ subject & \begin{bmatrix} agree & \begin{bmatrix} number & pl \\ person & 3 \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

We say that a feature structure F_1 subsumes another structure F_2 if all the attribute-value pairs in F_1 are also contained in F_2. For example, consider the following two structures.

$$[number \quad sg] \qquad \begin{bmatrix} person & 3 \\ number & sg \end{bmatrix}$$

The left feature structure subsumes the right, but not *vice versa*. Every attribute-value pair in the left structure is contained in the right, but the right structure contains an additional pair. The left structure can be thought of as being less specific than the right structure; this fits in with our intuitive notion of a less specific (more abstract) structure subsuming an equally or more specific one.

A unification grammar is formed by augmenting the production rules of a regular context-free grammar with constraints of the form

1. < X_i *feature path* > = *atomic value*
2. < X_i *feature path* > = < X_j *feature path* >

Given a production rule with terminal symbol X_i, the notation < X_i *feature path* > denotes a feature path through the feature structure associated with X_i. The constraints specify (1) that the value found *via* the given path must unify with the given atomic value, and (2) that the values found *via* the two given paths must be unifiable.

Consider our (extended) example context-free grammar with $N \equiv \{S, np, det, n, vp, iv\}$, $\Sigma \equiv \{a, all, brother, brothers, schemes, scheme\}$, starting symbol S, and the following production rules.

1. $S \rightarrow np\ vp$
2. $np \rightarrow det\ n$
3. $vp \rightarrow iv$
4. $det \rightarrow a$
5. $det \rightarrow all$
6. $n \rightarrow brother$
7. $n \rightarrow brothers$
8. $iv \rightarrow schemes$
9. $iv \rightarrow scheme$

Recall that we want to generate the strings *a brother schemes* and *all brothers scheme*, while disallowing *a brother scheme*, *all brother scheme*, *a brothers schemes* and *all brothers schemes*. We can do this by augmenting the production rule as follows.

$S \rightarrow np\ vp$

< *np number* > = < *vp number* >

This means that a sentence of the grammar may only be formed if the grammatical number of the noun phrase is equal to the grammatical number of the verb phrase. If we want to add a < *np person* > = < *vp person* > constraint to the rule – instead of listing the two constraints – we can make use of the *agree* attribute. We write

$S \rightarrow np\ vp$

< *np agree* > = < *vp agree* >

Using this new notation, our production rules are constrained as follows.

1. $S \rightarrow np\ vp$

   < *np agree* > = < *vp agree* >
2. $np \rightarrow det\ n$

   < *det agree* > = < *n agree* >

   < *np agree* > = < *vp agree* >
3. $vp \rightarrow iv$

   < *vp agree* > = < *iv agree* >
4. $det \rightarrow a$

   < *det agree* > = *sg*
5. $det \rightarrow all$

   < *det agree number*> = *pl*

6. *n* → *brother*
   < *n agree number*> = *sg*
7. *n* → *brothers*
   < *n agree number*> = *pl*
8. *iv* → *schemes*
   < *iv agree number*> = *sg*
   < *iv agree person*> = *3*
9. *iv* → *scheme*
   < *iv agree number*> = *pl*

It's worth pointing out that in a number of rules, a structure of a grammatical subcategory is copied into a parent category. The subcategory that provides the structure is usually referred to as the head of the phrase, whereas the structure copied is usually referred to as the head feature. Following our example, *n* is the head of the noun phrase and *iv* is the head of the verb phrase. In both cases *agree* is the head feature. We can rewrite our production rules to reflect these notions by placing the *agree* feature structure under a *head* attribute and copying that feature structure upwards.

1. *S* → *np vp*
   < *n head agree* > = < *vp head agree* >
2. *np* → *det n*
   < *det head agree* > = < *n head agree* >
   < *np head* > = < *n head* >
3. *vp* → *iv*
   < *vp head* > = < *iv head* >
4. *det* → *a*
   < *det head agree number*> = *sg*
5. *det* → *all*
   < *det head agree number*> = *pl*
6. *n* → *brother*
   < *n head agree number*> = *sg*
7. *n* → *brothers*
   < *n head agree number*> = *pl*
8. *iv* → *schemes*
   < *iv head agree number*> = *sg*
   < *iv head agree person*> = 3
9. *iv* → *scheme*
   < *iv head agree person*> = *pl*

3.3.1 Chart Parsing with Unification-Based Grammars

Recall from Section 3.2, that the edges of a chart can be described using the notation (i, j, L) where i is the start position of the edge, j is the end position and L is the label of the edge. In order to chart parse unification-based grammars, we add an additional field Fcontaining the feature structure associated with the label L.

The fundamental rule remains unchanged except for its handling of feature structures. The rule combines edges $(i, j, X \rightarrow \alpha \bullet Y \beta, F_1)$ and $(j, k, Y \rightarrow \gamma \bullet, F_2)$ to form the new edge $(i, k, X \rightarrow \alpha Y \bullet \beta, F_3)$. Here i, j and k are integers between 0and n; X and Y are non-terminal symbols; α is a (possibly empty) sequence of non-terminal symbols; β is a sequence of non-terminal symbols; γ is either a terminal symbol or a sequence of non-terminal symbols; and F_1, F_2, and F_3 are feature structures.

Essentially the structure F_3 is a version of F_1 whereby Y's feature structure in F_1 has been unified with Y's feature structure in F_2. It is important to note that a new edge is formed only if Y's feature structures in F_1 and $F_2$are unifiable, otherwise the rule is not applicable.

This new adaptation of the fundamental rule is best explained using an example. Consider the two constrained production rules from our grammar above.

1. $np \rightarrow det\ n$

   $< det\ head\ agree > = < n\ head\ agree >$

   $< np\ head > = < n\ head >$

2. $det \rightarrow a$

   $< det\ head\ agree\ number > = sg$

The structure F_1 of the edge $(0, 0, np \rightarrow \bullet\ det\ n, F_1)$ is built from the constraints of $np \rightarrow det\ n$. Top level attributes are created for each non-terminal symbol of the production rule. Hence we have

$$F_1 \equiv \begin{bmatrix} np & [head \otimes] \\ det & [head\ [agree \oplus [\ ]]] \\ n & [head \otimes [agree \oplus]] \end{bmatrix}$$

(Here $\oplus$ is used to indicate a re-entrant structure different from $\otimes$.) Similarly the structure $F_2$of the edge $(0, 1, det \rightarrow a \bullet F_2)$ is built from the constraints of $det \rightarrow a$. Here

$$F_2 \equiv [det\ [head\ [agree\ [number\ sg]]]]$$

We apply the fundamental rule, forming the new edge $(0, 1, np \rightarrow det \bullet n, F_3)$. The rule unifies the structures found under the *det* attribute in both F_1 and F_2. The structure F_3 is formed by taking F_1 and replacing the original structure under *det* by the newly unified structure. Hence we have

$$F_3 \equiv \begin{bmatrix} np & & [head \ \otimes] \\ det & [head & [agree \ \oplus [number \ \ sg]]] \\ n & & [head \ \otimes [agree \ \oplus]] \end{bmatrix}$$

A unification algorithm is described in detail in (Jurafsky and Martin 2000). To give a cursory description, the algorithm takes two feature structures represented as DAGs as input. The algorithm moves through the attributes (arcs) of one DAG and attempts to find a corresponding attribute in the other. If the attribute of one DAG is found to have no corresponding attribute in the other, the algorithm adds a directed arc to the deficient DAG pointing to the missing attribute. To keep the computational costs down – rather than construct a new DAG from scratch – the algorithm destructively alters the input DAGs to form the unified structure. Unification fails if any feature structures are found to be incompatible.

Recall – from Section 3.2.2 – that the general chart parsing algorithm for context-free grammars involves: (1) constructing an initial agenda and chart; (2) popping edges from the agenda and – as long as they are not already there – adding them to the chart; (3) applying the fundamental rule to edges of the chart, pushing any new edges on the agenda; and (4) hypothesising active edges – based on the current edge and the rules of the grammar – and pushing these new edges on the agenda.

The general algorithm works in much the same way for unification-based grammars. Apart from the changes to the fundamental rule we have already mentioned, there is one other difference. When parsing context-free grammars, we only add edges to the agenda that are not already present in the chart. When it comes to unification grammars, we only add edges that cannot be subsumed by edges already present in the chart. To see the reasoning behind this we will again look at an example.

Suppose we have a chart containing an edge $(0, 0, np \rightarrow \bullet \ det \ n, F_1)$ where F_1 places no restriction on *det*, *i.e.* the path < *det head agree number* > has the value [ ]. Consider an edge $(0, 0, np \rightarrow \bullet \ det \ n, F_2)$ where F_2 is the same as F_1, except that it restricts < *det head agree number* > to *sg*. Hence the latter edge is subsumed by the former. Consider the situations where

1. The parser adds $(0, 0, np \rightarrow \bullet \ det \ n, F_2)$ to the chart and encounters an edge $(0, 1, det \rightarrow a \ \bullet, F_3)$.
2. The parser adds $(0, 0, np \rightarrow \bullet \ det \ n, F_2)$ to the chart and encounters an edge $(0, 1, det \rightarrow all \ \bullet, F_4)$.

In the first situation, the fundamental rule is applicable to the pair of edges $(0, 0, np \rightarrow \bullet \ det \ n, F_1)$ and $(0, 1, det \rightarrow a \ \bullet, F_3)$ as well as being applicable to the pair $(0, 0, np \rightarrow \bullet \ det \ n, F_2)$ and $(0, 1, det \rightarrow a \ \bullet, F_3)$. Both applications will form identical edges $(0, 1, np \rightarrow det \ \bullet \ n, F_5)$ where F_5 specifies that < *det head agree number* > is singular. In the second situation, the fundamental rule is only applicable to the pair of edges $(0, 0, np \rightarrow \bullet \ det \ n, F1)$ and $(0, 1, det \rightarrow all \ \bullet, F_4)$.

Both situations suggest nothing worthwhile is achieved by adding the edge $(0, 0, np \rightarrow \bullet\ det\ n, F_2)$ to the chart in the first place; this is because there is a grammatically similar, but less constrained edge already in the chart. Adding an edge that is subsumed by another already in the chart just creates unnecessary work for the parser.

The definite clause grammar notation is used to implement PENG's unification-based grammar and chart parser in Prolog. Before discussing aspects of PENG's grammar and chart parsing techniques in Section 3.5, we'll provide some Prolog background material and look briefly at this grammar notation.

3.4 Logic Grammars

Logic grammars refer to grammars written in logic programming languages. Easily the most common logic grammar formalism is the Definite Clause Grammar (Sterling and Shapiro 1994). This grammar arises from adding features of the programming language Prolog[3] to context-free grammars. In order to describe the Definite Clause Grammar notation in more detail, we start with some Prolog fundamentals.

3.4.1 Prolog Basics

We should first point out that there are numerous implementations of Prolog: SWI Prolog, Strawberry Prolog, GNU Prolog, BProlog, *etc.* Our discussion here is based on the ISO standard Prolog language. See the reference manual (Deransart, Ed-Dbali et al. 1996) for details.

As discussed in (Gal, Lapalme et al. 1991; Blackburn, Bos et al. 2003) there are only three constructs in Prolog: facts, rules and queries. These constructs are built using terms. Terms are one of the following: an atom, a number, a variable, a complex term, or a list. Atoms are written in uncapitalised mixed case, whereas variables are written in capitalised mixed case. A complex term is an atom bracketing a sequence of one or more terms separated by commas. Some examples include `woman(_)`, `height(X)`, `age(42)` and `vp(iv(lives),pp(inDreadsburyMansion))`. Here `woman`, `height`, `age`, `vp`, `iv`, `lives`, `pp` and `inDreadsburyMansion` are all atoms; the atoms `woman`, `height`, `age`, `vp`, `iv` and `pp` acting as predicates; the lone underscore `_` of `woman(_)` is an anonymous variable which is left unspecified; `X` is a variable; and `42` is, well, a number. We will delay describing lists for a short while.

A Prolog program consists of a set of rules and facts. Rules are of the following form, where `p`, `q1`, `q2` and `q3` are all complex terms, nested or otherwise. (Note that `p` may be a nested complex term.

```
p :- q1,q2,q3.
```

This can be read as '`p` is true if `q1` and `q2` and `q3` are true'. The complex terms `q1`, `q2` and `q3` can be thought of as the conditions under which the complex term `p` is true. A fact is simply a rule with no conditions, *i.e.* `brother(charles).` It's worth pointing out that any variables

[3] Computational linguistics is a classic application for Prolog; the language's inventor, Alain Colmerauer, was a computational linguist.

featuring in the right-hand side of a rule do not necessarily have to appear in the left-hand side, *e.g.*

```
grandfather(X,Y) :- father(X,Z),father(Z,Y).
```

Moreover, variables in the left-hand side do not have to appear in the right, *e.g.*

```
append([A|X],Y,[A|Z]) :- append(X,Y,Z).
```

We will explain this last rule shortly.

A program is executed by initiating a query after the prompt `?-`. Prolog attempts to verify the query - using the existing rules and facts - and responds with an answer. For example suppose we have the following Prolog program.

```
woman(agatha).
dreary(dreadsburyMansion).
disinherited(charles).
gloomy(agatha) :- dreary(dreadsburyMansion).
schemes(charles) :- unhappy(charles).
unhappy(charles) :- disinherited(charles).
```

Prolog will respond to the various queries as follows.

```
?- woman(agatha).
yes

?- dreary(agatha).
no

?- woman(X).
X = agatha
yes

?- gloomy(agatha).
yes

?- schemes(charles).
yes
```

Lists in Prolog come in three varieties: empty, enumerated or head-tail lists. An empty list is represented by `[]`. In an enumerated list the elements are listed explicitly, *e.g.* `[`τ_1`, ... ,`τ_n`]`, where τ_i for $1 \le i \le n$ is a term. In a head-tail list `[`τ_1`|`τ_2`]` the element τ_1 is referred to as the head of the list, *i.e.* the first element; and τ_2 is the tail, *i.e.* the rest of the list. Here both τ_1 and τ_2 are terms. We give some example queries featuring head-tail lists below. Note that the empty list cannot be 'pulled apart' since it has no head.

```
?- [X|Y] = [a,b,c,d].
X = a
Y = [b,c,d]

?- [X|Y] = [].
no
```

The rule which we mentioned previously,

```
append([A|X],Y,[A|Z]) :- append(X,Y,Z).
```

stipulates that a head-tail list `[A|X]` appended to `Y` forms `[A|Z]` if `X` appended to `Y` forms `Z`.

A difference list in Prolog represents the difference between two lists and should not be confused with an actual list. Difference lists are of the form `[`τ`|X]-X` or γ_1`-`γ_2, where τ is a term, and γ_1 and γ_2 are either variables or the empty list. For example the difference list `[`τ_1`,` τ_2`|X]-X` represents `[`τ_1`,` τ_2`]` and `X-[]` represents `X`. The difference lists `X-X` and `[]-[]` both represent `[]`. The first list of a difference list is commonly referred to as the input list, the second is referred to as the output list. The use of difference lists leads to more concise and efficient programs, since appending difference lists is much simpler than appending standard lists. Difference lists can be appended in one step, whereas the number of steps needed to append two standard lists is equal to the length of the first list.

We can use difference lists when implementing a grammar recogniser; such a program allows us to check whether strings are grammatically correct with respect to a given grammar. (Note that a parser shares this feature, but – unlike a recogniser – it also provides us with information about the syntactic structure of the string.) For example, consider the following Prolog program.

```
s(X-Z) :- np(X-Y),vp(Y-Z).                    1
np(X-Z) :- det(X-Y),n(Y-Z).                   2
vp(X-Z) :- cop(X-Y),adj(Y-Z).                 3
det([the|W]-W).                               4
n([groundskeeper|W]-W).                       5
cop([is|W]-W).                                6
adj([drunk|W]-W).                             7
```

The rule at line 1 essentially says that a difference list `X-Z` is a sentence if (1) the difference between `X` and `Y` is a noun phrase; and (2) the difference between `Y` and `Z` is a verb phrase. The fact at line 4 says that the difference between `[the|W]` and `W` is the determiner `the`. We can query whether a difference list is either a sentence or a noun phrase as follows.

```
?- s([the,groundskeeper,is,drunk]-[]).
yes

?- np([the,groundskeeper]-[]).
yes
```

Feature structures in Prolog are represented as lists of attribute-value pairs. These pairs can be implemented following the method shown in (Blackburn and Striegnitz 2002). A colon is used to form the pairs; the attribute is to the left of the colon and the value is to the right. Attributes are represented by Prolog atoms. Values are atoms, variables, lists, difference lists, or are themselves attribute-value pairs. For example, the feature structure

$$\begin{bmatrix} \textit{category} & \textit{np} \\ \textit{number} & \textit{sg} \\ \textit{person} & 3 \end{bmatrix}$$

is represented in Prolog as `[cat:np,num:sg,pers:third]`. We write `cat`, `num` and `pers` instead of `category`, `number` and `person` since the notation is used by PENG. The nested structure

$$\begin{bmatrix} \textit{category} & \textit{np} & \\ \textit{agree} & \begin{bmatrix} \textit{number} & \textit{sg} \\ \textit{person} & 3 \end{bmatrix} & \end{bmatrix}$$

is represented as `[cat:np,agr:[num:sg,pers:third]]`. The re-entrant structure

$$\begin{bmatrix} agree & \otimes \\ subject & \begin{bmatrix} agree & \otimes \begin{bmatrix} person & 3 \\ number & sg \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

has the following representation

```
[agr:[pers:third,num:sg],subj:[agr:[pers:third,num:sg]]]
```

We won't go into too much detail regarding Prolog unification here; instead we refer the reader to (Brett 2000). We will mention however that list unification is recursive. The first elements of both lists are compared, and if they unify, then the second elements are compared, and so on. As discussed in (Blackburn and Striegnitz 2002) variations to the standard unification algorithm can be easily implemented. For example, an algorithm can be implemented such that structures `[cat:np,num:sg]` and `[cat:np]` are unified to `[cat:np,num:sg]`, and structures `[cat:np]` and `[num:sg]` are unified to `[cat:np,num:sg]`. Moreover, we can implement the algorithm such that it unifies structures of different attribute orderings. For example we can unify `[cat:np,num:sg]` and `[num:sg,cat:np]` to `[cat:np,num:sg]`. We point this out since PENG implements such a variation to the standard algorithm.

3.4.2 Definite Clause Grammars

The Definite Clause Grammar (DCG) notation is implemented in most versions of Prolog – including the ISO standard language – and acts as syntactic sugar for rules and facts featuring difference lists. Using this notation we can avoid having to keep track of all the difference list variables. We can rewrite the above Prolog program in DCG notation as follows.

```
s --> np,vp.
np --> det,n.
vp --> cop,adj.
det --> [the].
n --> [groundskeeper].
cop --> [is].
adj --> [drunk].
```

Prolog simply translates these DCG rules into the rules and facts of our original program. The program is queried in the same manner as before.

Since unification can be implemented in Prolog, it is possible to add feature structures to the DCG rules. For example, consider the following constrained production rule written in the notation of Section 3.3.

$$S \rightarrow np\ vp$$
$$< np\ number > = < vp\ number >$$
$$< np\ person > = < vp\ person >$$

This states that a sentence of the grammar may only be formed if the grammatical number of the noun phrase agrees with the number of the verb phrase, and the grammatical person of the noun phrase agrees with the person of the verb phrase. Such a rule can be represented in DCG notation as

```
s --> np(num:N,pers:P),vp(num:N,pers:P).
```

We can desugar the DCG rule into an ordinary Prolog rule. Namely

```
s(X-Z) :- np(num:N,pers:P,X-Y),vp(num:N,pers:P,Y-Z).
```

Similarly, the production rule

det → *the*

< *det agree number* > = [ ]

can be represented in DCG notation as

```
det(agr:num:_) --> [the].
```

This DCG rule can be desugared into the fact

```
det([agr:num:_,the|W]-W).
```

We saw previously how to implement a grammar recogniser in Prolog; we'll now look at implementing a parser. First of all consider a nested complex term such as `s(np(det(the),n(groundskeeper)),vp(cop(is),adj(drunk)))`. This term captures the structure of the following tree.

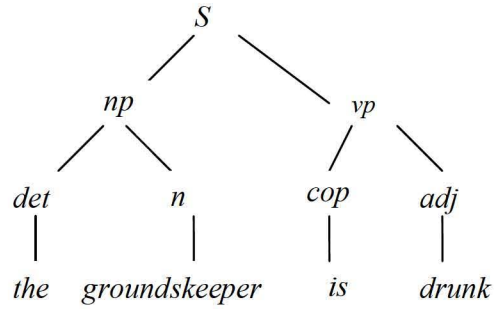


Figure 8: Structure of a complex term

In order to build such complex terms, we simply add extra arguments to the rules. Consider, for example

```
sentence(s(NP,VP)) --> nounPhrase(NP),verbPhrase(VP).
nounPhrase(np(DET,N)) --> determiner(DET),noun(N).
verbPhrase(vp(COP,ADJ)) --> copula(COP),adjective(ADJ).
determiner(det(the)) --> [the].
noun(n(groundskeeper)) --> [groundskeeper].
copula(cop(is)) --> [is].
adjective(adj(drunk)) --> [drunk].
```

The grammar builds the parse tree for the grammatical categories on the left-hand side of the rules out of the grammatical categories on the right. The extra argument `s(NP,VP)` in the rule `sentence(s(NP,VP)) --> nounPhrase(NP),verbPhrase(VP)` forms a term whose predicate is `sentence` and whose first and second arguments are the values of `NP` and `VP` respectively.

The following query asks Prolog to instantiate `T` with the parse tree for the sentence.

```
?- sentence(T,[the,groundskeeper,is,drunk]-[]).
```

Prolog replies

```
T = s(np(det(the),n(groundskeeper)),vp(cop(is),adj(drunk)))
yes
```

A query such as `sentence(T,S-[])` returns all parse trees.

A further benefit of the DCG notation is that it allows the user to embed Prolog code and hence separate out the DCG rules and lexicon. Following our example, we have the lexicon

```
lex(the,det).
lex(groundskeeper,n).
lex(is,cop).
lex(drunk,adj).
```

along with the DCG rules

```
s --> np,vp.
np --> det,n.
vp --> cop,adj.
det --> [W],{lex(W,det)}.
n --> [W],{lex(W,n)}.
cop --> [W],{lex(W,cop)}.
adj --> [W],{lex(W,adj)}.
```

The `det` rule – with its embedded code in curly brackets – tells us that a determiner can consist of a list containing a single element `W` as long as `W` is a determiner within the lexicon. The `n`, `cop` and `adj` rules function similarly.

PENG not only implements a separate lexicon and set of DCG rules, it also employs gap threading. Before looking at aspects of the PENG grammar, we discuss this technique in the next section.

3.4.3 Gap Threading in Definite Clause Grammars

Gap threading is a technique that – when applied to a context-free grammar – rules out grammatically incorrect relative clauses. The following description follows from (Blackburn and Striegnitz 2002). See also (Nugues 2006) and (Meurers 2003) for more discussion. Recall that a relative clause is a clause that modifies a noun. For example *rc* labels the relative clauses in the following strings.

1. *the groundskeeper (who Agatha likes)*$_{rc}$
2. *Agatha, (who likes the groundskeeper)*$_{rc}$

It is worth noting that we can form these noun phrases from the string *Agatha likes the groundskeeper*. Here *Agatha* is the subject of the sentence and *the groundskeeper* is the object. To form String 1 we perform object relativisation: we (1) extract *the groundskeeper* from its original position and move it to the front of the string leaving a 'noun-phrase gap', and (2) insert the relative pronoun *who* between *the groundskeeper* and the remainder of the string. Hence we end up with the string *the groundskeeper who Agatha likes* [*gap*]. To form the String 2 we perform subject relativisation: we (1) force a gap between *Agatha* and the rest of the string, and (2) insert *who* between the gap and the remainder of the string. Hence we end up with *Agatha,* [*gap*] *who likes the groundskeeper*.

Assuming our context-free grammar contains production rules that account for complete sentences, we can use the notion of gaps to capture the almost-sentence characteristics of relative clauses. We'll show how this is done *via* an example. Suppose we have the following DCG rules.

```
s --> np,vp.
np --> pn.
np --> det,n.
vp --> tv,np.
pn --> [agatha].
det --> [the].
n --> [groundskeeper].
tv --> [likes].
```

Using this grammar we can generate lists of the form `[np,likes,np]`, where `np` is either the proper noun `agatha` or the noun phrase `[the,groundskeeper]`. Adding the following four rules allows us to relativise the subject.

```
np --> det,n,rc.
np --> pn,rc.
rc --> relpro,vp.
relpro --> [who].
```

We can generate lists of the form `[np,likes,np]` and `[np,who,likes,np]`, where `np` is `agatha`, `[the,groundskeeper]` or `[np,who,likes,np]`. These generated lists represent grammatically correct sentences – albeit a little on the repetitive side – but we still can't build sentences of the form `[np,who,np,likes]`. If we add the rule

```
rc --> relpro,s.
```

we can generate a list of the form `[np,who,np,likes,np]`, where `np` is `agatha`, `[the,groundskeeper]` or `[np,who,likes,np]`. However the sentences these lists represent don't make much sense, *e.g.* `[agatha,who,the,groundskeeper,likes, the,groundskeeper]`. In order to relativise the object, we need a way of deleting the last noun phrase of the list; we do this by introducing gap feature structures.

We add the following rule to our DCG.

```
np(gap) --> [].
```

This rule means that an 'empty' `np` can be used, as long as the noun phrase is constrained by a `gap` atom. We rewrite the grammar as follows.

```
s(G) --> np(nogap),vp(G).
np(nogap) --> pn.
np(nogap) --> det,n.
np(nogap) --> det,n,rc.
np(nogap) --> pn,rc.
np(gap) --> [].
rc --> relpro,vp(nogap).
rc --> relpro,s(gap).
vp(G) --> tv,np(G).
pn --> [agatha].
det --> [the].
n --> [groundskeeper].
relpro --> [who].
tv --> [likes].
```

The grammar is constructed such that the value of the `G` variable constraining the `s` rule is either `gap` or `nogap` depending on whether the verb phrase contains an empty noun phrase. Note also that a sentence cannot consist of an empty noun phrase followed by a verb phrase; this is not allowed in English. This DCG allows us to generate the desired strings, but it's still rather limited.

Consider a string such as *Charles tells the police about the groundskeeper*. We can perform object relativisation in three different ways: (1) we can form the string *the groundskeeper who Charles tells the police about* [*gap*]; (2) we can form the string *the police who Charles tells* [*gap*] *about the groundskeeper*; and (3) we can re-relativise string 2, forming *the groundskeeper who the police who Charles tells* [*gap*] *about* [*gap*]. Obviously, string 3 doesn't make much sense. In some languages you can perform multiple extractions/relativisations, but in English you can't. Hence we need to factor this new constraint into our grammar; we do this by 'threading' gaps through difference lists. Consider the following DCG.

```
s(F-G) --> np(F-F),vp(F-G).
np(F-F) --> pn.
np(F-F) --> det,n.
np(F-F) --> det,n,rc.
np([gap|F]-F) --> [].
rc --> relpro,s([gap|F]-F).
vp(F-G) --> dv,np(F-H),pp(H-G).
pp(F-G) --> p,np(F-G).
pn --> [charles].
det --> [the].
n --> [groundskeeper].
n --> [police].
dv --> [tells].
relpro --> [who].
p --> [about].
```

Recall that a difference list represents the difference between the contents of the input and output lists. A rule such as

```
np(F-F) --> pn.
```

tells us that a noun phrase can consist of a proper noun containing no gaps: the input list is the same as the output list. In the DCG, the difference list `F-F` is essentially playing the role of `nogap`. The rule

```
np([gap|F]-F) --> [].
```

tells us that a noun phrase can be empty, and if it is, we need to add the atom `gap` to the beginning of the input list. In this case, the difference between the input and output lists is precisely `gap`. The rule

```
rc --> relpro,s([gap|F]-F).
```

tells us that a relative clause contains a single gap, whereas

```
pp(F-G) --> p,np(F-G).
```

tells us that a prepositional phrase may be empty depending on whether the noun phrase is empty. Hence a gap can be passed 'up' from the noun phrase to the prepositional phrase. Finally, the rule

```
vp(F-G) --> dv,np(F-H),pp(H-G).
```

tells us that it is possible for a verb phrase to contain multiple gaps if both the noun and prepositional phrases contain gaps. Note however that within a relative clause, a verb phrase

contains a single gap. In this case, there must be either a gap in the noun phrase or a gap in the prepositional phrase, but no gap in both. If the difference between `F` and `G` is `gap` then either (1) the difference between `F` and `H` is `gap` and thus the difference between `H` and `G` is `[]`; or (2) the difference between `F` and `H` is `[]` and thus the difference between `H` and `G` is `gap`.

It's worth mentioning that this example grammar is not particularly good since it allows us to construct full, ungrammatical sentences such as *Charles tells about*, *Charles the the groundskeeper about* and *Charles tells about the groundskeeper*. It is possible to exclude such sentences by constraining the grammar such that verb phrases may only contain gaps when in relative clauses. However to avoid complicating matters, we will not go into further detail here.

3.5 The PENG Grammar

In this section we provide a description of a number of grammar rules in all their feature-laden glory, before concluding the section with a discussion of PENG's incremental chart parsing techniques.

3.5.1 Grammar Rule Examples

As discussed in Section 2.1, PENG's base lexicon consists of: predefined function words including determiners, connectives, prepositions and relative pronouns; and approximately 3,000 predefined content words, including nouns, proper nouns, verbs, adjectives and adverbs.

Below is a typical PENG lexical entry; the entry for the word *butler*.

```
lexicon([lex:[butler],synon:[pantryman]],
        [cat:cn,
         arg:[ind:I,type:person,agr:[per:third,num:sg,gend:_],
              case:_],
         con:[obj([butler],I),struc(I,atomic)]],base).
```

We'll work through the constraint list backwards. The atom `base` is used to distinguish lexical entries that belong to the base lexicon from those that are user-defined.

The value `[obj([butler],I),struc(I,atomic)]` of attribute `con` is a list of conditions used for the construction of a discourse representation structure. This process is discussed in detail in Section 3.6.2. For now, the word *butler* can be thought of as a concept and variable `I` can be thought of as an object that falls under this concept. The structure of the object `I` is specified as being atomic, *i.e.* it is an individual object, rather than, say, a member of a group.

The argument attribute `arg` takes as its value a list of feature structures. The attribute-value pair `case:_` specifies that *butler* has either nominative or accusative case. The attribute-value pair `agr:[per:third,num:sg,gend:_]` specifies agreement properties; third person, singular grammatical number and non-specific gender. The attribute `type` is used to distinguish between, say, the sentences *Agatha digs a grave* and *Agatha digs the butler*. In the former sentence, the word *grave* has type `entity`; whereas in the latter, the word *butler* has type `person`. The attribute-value pair `ind:I` specifies that the value for the 'index' attribute `ind` is the variable `I`; this specification should not be confused with sequential indexing.

The attribute-value pair `cat:cn` categorises *butler* as a common noun, whereas the pair `synon:[pantryman]` indicates the word *pantryman* can be used as a synonym.

PENG's unification-based grammar currently has about 150 production rules. Below is the simplified DCG rule for the common noun `n0` (Schwitter 2004a).

```
n0([cat:cn,
    arg:[ind:I,type:T,agr:A|Rest],
    ...])
    -->
   {lexicon([lex:Noun],
            [cat:cn,
             arg:[ind:I,type:T,agr:A|Rest],
             con:[C3,C2]]},
    Noun.
```

Hence when PENG's chart parser parses a word such as *butler*, the value of the `lex` attribute of the lexical entry (the list `[butler]`) unifies with the `lex` attribute of the `n0` rule (the variable `Noun`). Moreover, the conditions `obj([butler],I) and struc(I,atomic)]` unify with the variables `C3` and `C2`. These conditions feature as values of a `drs` attribute usually found on the left-hand side of the rule. We have not included this attribute here since it is somewhat complicated; we discuss it later in Section 3.6.2. The `arg` feature structure of the lexical entry unifies with the `arg` feature structure on the left-hand side of the DCG rule. The resultant feature structure `arg:[ind:I,type:person,agr:[per:third,num:sg,gend:_],case:_|Rest]` is then copied to the right-hand side of the rule.

Below is another simplified example DCG rule which shows that the noun phrase `n3` is composed of a determiner `d0` and noun `n2` (Schwitter 2007a).

```
n3([coord:no,
    arg:[ind:I,type:T,agr:A|Rest],
    spec:def,
    ana:yes,
    para:P1-P4,
    tree:[n3,T1,T2],
    gap:n3:G-G,
    styp:Y,
    snum:N,
    ...])
    -->
    d0([cat:det,
        agr:A,
        spec:def,
        ana:yes,
        para:P1-P2,
        tree:T1,
        snum:N,
        ...]),
    n2([cat:cn,
       arg:[ind:I,type:T,agr:A|Rest],
       spec:def,
```

```
        para:P2-P3,
        tree:T2,
        gap:n3:G-G,
        styp:Y,
        snum:N,
        ...]),
    {anaphora_resolution(n3,cn,I,...)}.
```

Note that we are about to describe a number of attributes – `coord`, `spec`, `ana`, `para`, `tree`, `styp`, `snum` – which will not feature in the sequel; we only include them here to give a sample of the grammar rules.

The `coord:no` attribute-value pair indicates that a coordinating conjunction at the `n3` noun phrase level is not allowed. The `arg` feature structures of the rule indicate that the arguments of noun `n2` are copied upwards to the noun phrase `n3`. Note the `d0` and `n2` agreement over variable `A`. The attribute-value pair `spec:def` specifies `n2`, `d0` and `n3` as definite articles, *cf.* indefinite articles (*e.g. a, some* and *sombody*) or quantifiers. The attribute-value pair `ana:yes` indicates that the noun phrase may be used anaphorically. Since a definite noun phrase can be used anaphorically in PENG – along with proper nouns and variables – the lexical entries for definite determiners are unifiable with `ana:yes` attribute-value pairs.

The attribute `para` constructs a paraphrase for the input string, whereas the attribute `tree` builds a parse tree during processing time. The attribute `gap` takes care of gap threading. The attribute `styp` refers to sentence type, *i.e.* whether a sentence is a declarative statement or an interrogative question. The grammar is constructed such that the variable `Y` will unify with either `decl` or `int`. The attribute `snum` refers to the sentence number, an integer that identifies the sentence.

The Prolog code in curly brackets tests whether `n3` is used anaphorically. The predicate `anaphora_resolution` is triggered when the entire noun phrase has been processed. We discuss anaphora resolution in more detail in Section 3.6.2. In this later section we also discuss a number of attributes absent from the above rule; these attributes are used to build the related discourse representation structure.

In the following section we take a look at PENG's chart parsing approach and also see how the parser generates look-ahead categories on-the-fly.

3.5.2 Incremental Chart Parsing in PENG

PENG represents an edge of a chart using the notation *edge*(*ID*, v_i, v_t, *LHS*, *RHSL*) (Schwitter 2003). Here *ID* is an integer which acts as a sentence identifier, v_i and v_t are integers between 0 and n, where n is the number of constituents of the input string; *LHS* is the non-terminal symbol (*i.e.* grammatical category) on the left-hand side of the dotted rule labelling the edge; and *RHSL* represents the right-hand side of the rule in list form. If *RHSL* is empty, then the edge is inactive, otherwise the edge is active. For example an edge (0, 0, $S \rightarrow \bullet$ *np vp*) written in our usual format can be represented in PENG notation as `edge(0,0,s,[np,vp])`. (Note that we are neglecting the sentence identifier, as well as the feature structures constraining the grammatical categories of the relevant PENG DCG rule.)

Edges (0, 2, *np* → *det n* •) and (0, 2, *S* → • *np* • *vp*) can be represented as `edge(0,2,np,[])` and `edge(0,2,s,[vp])` respectively.

In Section 3.2.2 we showed how a top-down chart parser constructed a chart for the input string *the butler acts suspiciously*. Using the new PENG notation, the final chart contained the following edges:

```
edge(0,0,s,[np,vp])
edge(0,0,np,[det,n])
edge(0,1,np,[n])
edge(0,2,np,[])
edge(0,2,s,[vp])
edge(2,2,vp,[iv,adv])
edge(2,3,vp,[adv])
edge(2,4,vp,[])
edge(0,4,s,[])
edge(0,0,np,[det,n,rc])
edge(0,1,np,[n,rc])
edge(0,2,np,[rc])
edge(2,2,rc,[relpro,vp])
```

Without having to re-process the entire sentence, the parser – as we have described it – is unable to handle modifications to the input string. In contrast, an incremental chart parser *can* handle modifications such as insertion, deletion and replacement, without having to re-process the entire string from scratch. Such a parser uses information about edge dependencies for keeping track of edges that have to be updated (Wirén 1989; Wirén 1994).

In the sequel we follow the description outlined in (Schwitter 2003) and show how PENG's incremental top-down chart parser handles the modification of the input string *the butler acts suspiciously*. The final chart for the string is presented below. For readability we ignore the inactive edges `edge(0,1,det,[])`, `edge(1,2,n,[])`, `edge(2,3,v,[])`, and `edge(3,4,adj,[])`.

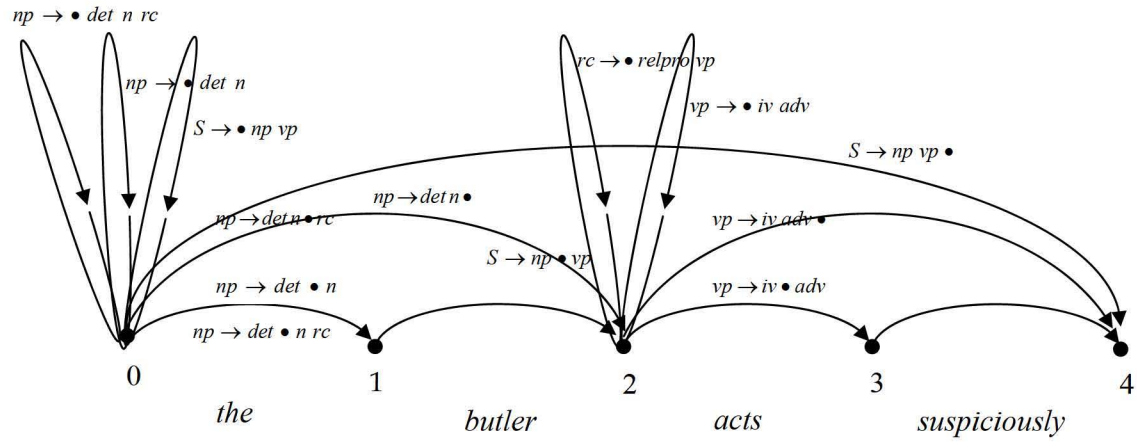


Figure 9: Parse Chart

Suppose we insert the relative pronoun *that* between the noun and verb phrase. The PENG incremental parser informally performs the following steps.

1. Find all edges on the right-hand side of the insertion point 2, *i.e.* all those edges with start point greater than or equal to the insertion point, and create a new subchart *CR* for them. Do not include the empty edge at the insertion point featuring the grammatical category of the word to be inserted, *i.e.* `edge(2,2,rc,[relpro,vp]).`
2. For all edges in *CR*, renumber their start and end points v_{s+1} and v_{t+1}.
3. Find all non-empty edges on the left-hand side of the insertion point, *i.e.* all those edges with start point less than the insertion point and end point less than or equal to the insertion point, and create a new subchart *CL* for them. Include in *CL* `edge(2,2,rc,[relpro,vp]).`
4. Create a new chart *C* by appending *CR* to the end of *CL*
5. Create new hypotheses – in the form of active edges – beginning at the insertion point for the word *that*.
6. Reparse the string using only the new edges in the agenda and the new chart *C*.

Note that this algorithm removes `edge(0,5,s,[])` from the chart. This is the only edge that spans the node where the word *that* would be inserted. Having followed the algorithm, the parser generates four new edges.

```
edge(2,3,rc,[vp])
edge(2,5,rc,[])
edge(0,5,np,[])
edge(5,5,vp,[iv,adv])
```

The parser also modifies the following four edges (modifications are in bold face).

```
edge(3,3,vp,[iv,adv])
edge(3,4,vp,[adv])
edge(3,5,vp,[])
edge(0,5,s,[vp])
```

PENG's chart parser implements similar algorithms for deletion and replacement operations. Another function of the parser is that it dynamically generates look-ahead categories for each word form. This guides the author and guarantees compliance to the rules of the controlled language. As defined in (Schwitter 2003), a set of look-ahead categories *LC* for a word *w* ending at node v_i can be found by following the procedure below.

1. Find all active edges ending at v_i.
2. For each active edge select the *RHSL*.
3. For the first category in *RHSL* check if it is a lexical category or non-terminal symbol.
    a. If yes, store the category in *LC*.
    b. If no, find a rule which rewrites the category into further subcategories, then select the first category and return to 3.

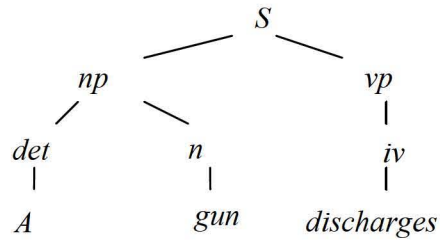
(Note that grammatical categories can be further subdivided into lexical categories, *i.e. det*, *n* and *iv*, and phrasal categories, *i.e. np*, *vp* and *pp*.) As each word form is entered into the text field of PENG's text editor ECOLE, the parser generates the set of look-ahead categories and sends them to the ECOLE editor.

3.6 Discourse Representation

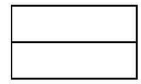
As mentioned previously, PENG's chart parser translates input text into a Discourse Representation Structure (DRS). A DRS is then translated into formulae of first-order logic which can be checked for consistency, informativity and/or used for question-answering by third-party reasoning services. In this section, we give a brief introduction to discourse representation theory and informally describe the standard DRS construction algorithm. In Section 3.6.1 we show how a DRS can be translated into first-order logic. Section 3.6.2 describes how PENG's chart parser constructs DRSs on-the-fly. The following descriptions of discourse representation is derived from (Blackburn and Bos 1999).

Consider the discourse *A gun discharges. The gun falls* (by discourse we mean a sequence of natural language sentences). In order to capture the meaning of this discourse, we could try a first-order representation such as $\exists x.\, Gun(x) \wedge Discharges\,(x) \wedge Falls(x)$. (Note that in first-order representations we are not permitted to quantify over predicates; such quantifications are allowable in higher-order representations.) We can see that $\exists x.\, Gun(x) \wedge Discharges\,(x) \wedge Falls(x)$ captures the fact that *The gun* refers back to *A gun*. In linguistic terms, the anaphor is said to have been resolved. Anaphora is an instance of one expression referring back to another (Mitkov 2003; Burchardt, Walter et al. 2005). In our example, the pronoun *The gun* is an anaphor, *A gun* is the antecedent, and *A gun* and *The gun* are said to be co-referential. (Note that often anaphors are personal pronouns such as *He*, *She*, *They* and *It*; since these are illegal in PENG, we will not feature them in our discussion here.) Importantly, our first-order representation does not capture how the discourse works. We are first told that a gun discharges; the subsequent discourse tells us a fact about the gun, namely that it falls. Essentially, the sentence *A gun discharges* changes the context in which subsequent discourse is interpreted. The first-order representation fails to capture this change of context. It puts the two facts – the gun discharging and falling – on equal footing, which does not entirely accord with our understanding of the text.

In Discourse Representation Theory (DRT), the meaning of a sentence is defined by how it can change contextually. Structures are built on-the-fly, providing a more accurate reflection of contextual change within discourse. Hans Kamp and Uwe Reyle's standard DRS construction algorithm is outlined in (Kamp and Reyle 1993; Blackburn and Bos 1999). We will only provide an informal description here. The algorithm begins by receiving the first sentence and works around the parse tree of that sentence in a top-down, left-to-right approach. As an example, consider the construction of the DRS for the discourse *A gun discharges. The gun falls.* For the first sentence *A gun discharges* we have the following parse tree.



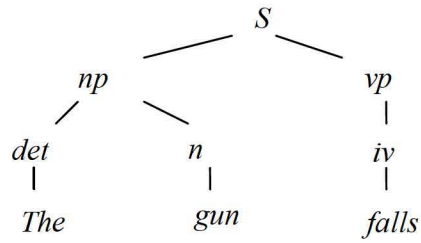
At the top of the parse tree the construction algorithm creates an empty DRS.



The algorithm then works down from *S* to the *np* node. Upon identifying the existentially quantified noun phrase a variable x is added to the top section of the DRS. Such a variable is called a discourse referent. The top section of the DRS is called the universe. Moving around the tree to the *n* node, the word *gun* is found. The algorithm places the expression $Gun(x)$ in the bottom section of the DRS. The expression is termed a condition and has the meaning that referent x is constrained by the concept *Gun*. The algorithm then moves back up to *S* and then down to the *vp* sub-tree. Here the verb phrase consists of the intransitive verb *discharges*. A further constraint is made on x and the condition $Discharges(x)$ is added to the bottom section of the DRS. Thus far the algorithm has constructed the following DRS.

X
$Gun(x)$
$Discharges(x)$

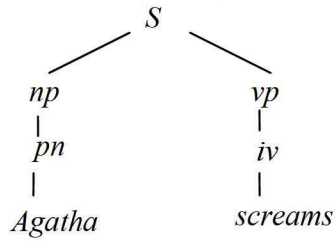
The second sentence *The gun falls* generates the following parse tree.



The algorithm adds the information obtained from the second sentence to the DRS already constructed. It moves from node *S* to node *np*. Upon identifying the noun phrase, the algorithm adds a new discourse referent y to the universe of the DRS. Moreover, since the noun of the noun phrase has been encountered before, the algorithm treats the word as anaphoric and adds the condition $y = ?$ to the bottom section of the DRS. The question mark must be identifiable with an accessible referent. We will discuss the accessibility of referents shortly. Since there is only one accessible discourse referent available, namely x, the algorithm substitutes ? with x. Essentially, the condition $y = x$ resolves the pronoun *The gun* with its antecedent *A gun*. The algorithm then works around to the *vp* sub-tree. Upon identifying the intransitive verb *falls*, it adds the condition $Falls(y)$ to the bottom section of the DRS. We then have the final structure.

$x \quad y$
$Gun(x)$
$Discharges(x)$
$y = x$
$Falls(y)$

The algorithm handles proper nouns in much the same way as it handles existentially quantified noun phrases. Consider the following parse tree.



The algorithm builds the DRS

X
x = agatha
Screams(x)

The noun phrase introduces the discourse referent x and identifies it with the constant *agatha*.

A vocabulary of a DRS language is comprised of

A unique set of predicate symbols of arity n such that $n \geq 1$. These symbols are denoted using capitalised mixed case, or more generally using P, Q and R.

A unique set of constant symbols. These symbols are denoted using uncapitalised mixed case, or more generally using a, b and c.

A unique set of function symbols of arity m such that $m \geq 1$. These are denoted using uncapitalised mixed case, or more generally using f, g and h.

Given a particular vocabulary, we build a DRS language over that vocabulary together with the following elements.

A finite set of discourse referents, denoted using x, y and z with subscripts.

The connectives $\neg$ (not), $\vee$ (or) and $\Rightarrow$ (implication).

Left and right parenthesis and the comma.

The equality symbol =.

A term is a constant, a discourse referent, or a function of m terms where $m \geq 1$. A primitive condition is a predicate of n terms where $n \geq 1$. If τ_1 and τ_2 are terms, then $\tau_1 = \tau_2$ is also a primitive condition. Formally, a DRS condition is defined as follows.

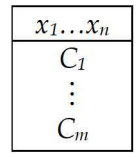
A primitive condition is a DRS condition.

If B_1 and B_2 are DRSs, then $B_1 \Rightarrow B_2$ and $B_1 \vee B_2$ are DRS conditions.

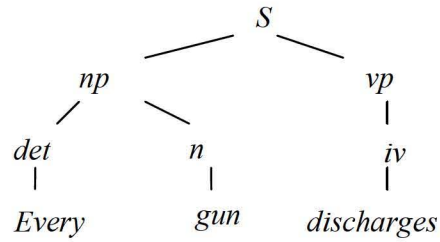
If B is a DRS, then $\neg B$ is a DRS condition.

Nothing else is a DRS condition.

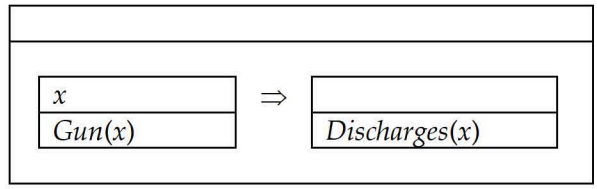
We now can formally define a DRS. If $x_1,\ldots,x_n$ are discourse referents and $C_1,\ldots,C_m$ are DRS conditions where both $n \geq 1$ and $m \geq 0$, then the following is a DRS.



We will delay discussing the semantics of the DRSs B, $B_1 \Rightarrow B_2$, $B_1 \vee B_2$ and $\neg B$ until the next section. For now, consider the parse tree below.



In order to build the DRS for this sentence, the algorithm creates an empty DRS at S and works down to the *np* node. Upon identifying the universally quantified noun phrase, the algorithm embeds a condition $B_1 \Rightarrow B_2$ in the bottom section of the empty DRS. Here B_1 is a DRS with no conditions and the sole discourse referent x and B_2 is an empty DRS. After encountering the word *gun* at node n, the algorithm adds the condition $Gun(x)$ to DRS B_1. The algorithm then moves back up to S and down to the *vp* sub-tree where it encounters the intransitive verb *discharges*. Here it adds the condition $Discharges(x)$ to DRS B_2. We have the following DRS.

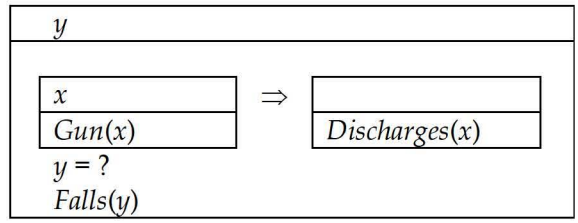


In a previous example, the standard construction algorithm added a new discourse referent y and the condition y = ? to a DRS when it came across a pronoun. According to the rules of DRS construction, the question mark must be identifiable with an accessible discourse referent. We say that a DRS B_1 is accessible from DRS B_2 when either B_1 is identical to B_2, or B_1 subordinates B_2. A DRS B_1 subordinates a DRS B_2 if and only if one of the following items holds.

1. B_1 contains a DRS condition of the form $\neg B$.
2. B_1 contains a DRS condition of the form $B_2 \Rightarrow B$ for some DRS B.
3. $B_1 \Rightarrow B_2$ is a DRS condition of some DRS B.
4. B_1 contains a DRS condition of the form $B_2 \vee B$ or $B \vee B_2$ for some DRS B.
5. There is some DRS B such that B_1 subordinates B and B subordinates B_2.

Intuitively, B_1 is accessible from B_2 if either $B_1$and B_2 are the same DRS, or if B_2 is nested within B_1.

We say that a discourse referent x belonging to a DRS B_1 is accessible from a discourse referent y belonging to a DRS B_2 if and only if B_1 is accessible from B_2. For example, suppose we construct a DRS for the discourse *Every gun discharges. The gun falls*. We have already seen the DRS built for the first sentence *Every gun discharges*. As the algorithm works it way through the parse tree of the second sentence, it identifies the noun phrase *The gun* and adds a new discourse referent y to the universe of the DRS. Since the noun of the noun phrase has been encountered before, the algorithm adds the condition $y = ?$ to the DRS. Since there is no accessible discourse referent from y – the only possible candidate is x and it is not accessible from y – then there is no correct anaphoric interpretation for *The gun*. Lastly, the algorithm identifies the verb *falls* and adds the condition $Falls(y)$ to the DRS. We have the final structure.



3.6.1 Discourse Representation and First-Order Logic

In this section we present a DRS translation to First-Order Logic (FOL). Before describing the translation, we give a quick recap of FOL from a model-theoretic standpoint. We also discuss a dynamic semantics for DRSs. Much of the section follows from (Blackburn and Bos 1999; Blackburn and Bos 2005).

A vocabulary of a first-order language is comprised of

A unique set of predicate symbols of arity n such that $n \geq 1$. These symbols are denoted using capitalised mixed case, or more generally using P, Q and R.

A unique set of constant symbols. These symbols are denoted using uncapitalised mixed case, or more generally using a, b and c.

A unique set of function symbols of arity m such that $m \geq 1$. These are denoted using uncapitalised mixed case, or more generally using f, g and h.

Informally, a vocabulary tells us two things: what we're going to talk about and how we're going to talk about it. For example suppose we have the constant symbol *charles* and the unary predicate symbol *Corpse* in our vocabulary. Then we are able to talk about the individual Charles and/or the property of being a corpse. Furthermore, we use the symbol *charles* to refer to Charles and *Corpse* to refer to any corpses.

Given a particular vocabulary, we build a first-order language over that vocabulary together with the following elements.

An infinite set of variable symbols, denoted using x, y and z with subscripts.

The connectives $\neg$ (not), $\wedge$ (and), $\vee$ (or) and $\Rightarrow$ (implication).

The quantifiers $\forall$ (universal) and $\exists$ (existential).

Left and right parenthesis and the comma.

Usually an equality symbol =.

A term of a first-order language is a constant symbol or a variable. Moreover, if f is a function symbol of arity m and $\tau_1,\ldots,\tau_m$ are terms, then $f(\tau_1,\ldots,\tau_m)$ is also a term. If P is a predicate symbol of arity n and $\tau_1,\ldots,\tau_n$ are terms then $P(\tau_1,\ldots,\tau_n)$ is said to be an atomic formula. If the equality symbol = is considered part of the language and if τ_1 and τ_2 are terms, then $\tau_1 = \tau_2$ is also an atomic formula. A well-formed formula (or WFF, or simply 'a formula') is defined as follows.

An atomic formula is a WFF.

If φ and ψ are WFFs, then $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \Rightarrow \psi$ are WFFs.

If φ is a WFF and x is a variable, then $\forall x.\varphi$ and $\exists x.\varphi$ are WFFs.

Nothing else is a WFF.

A formula can be thought of as a description. There are two more points worth noting about first-order formulae. First, quantification is permitted only over variables; this is what distinguishes FOL from higher-order logic. Second, a variable occurrence is said to be bound in a formula if it lies within the scope of a quantifier, otherwise is it said to be free.

Essentially, terms can be thought of as first-order versions of noun phrases: constants can be thought of as first-order versions of proper names, whereas variables can be thought of as pronouns. In natural language terms, an atomic formula corresponds to a sentence without the conjunctions *and* or *or*. A formula built using $\neg$ corresponds to an expression *It is not the case that…* A formula built using $\wedge$ corresponds to an expression *…and…*, whereas a formula built using $\vee$ corresponds to *…or…* A formula using $\Rightarrow$ corresponds to the expression *If…then…* Formulae of the form $\forall x.\varphi$ correspond to *All…* or *Every…*, whereas formulae of the form $\exists x.\varphi$ correspond to *There is a…*

A model for a given vocabulary can be thought of as a situation. Formally, a model M for a given vocabulary is a pair (D,F) specifying a non-empty domain D and an interpretation function F. The domain contains the kinds of things we want to talk about, *e.g.* individuals, places or objects. The interpretation function specifies for each symbol in the vocabulary a semantic value in the domain. Essentially, it provides an 'interpretation' for each symbol in the vocabulary. Each constant symbol a is interpreted as an element of the domain, *i.e.* $F(a) \in D$. For example $F(agatha)$ is some element of D, which we can specify as somebody called Agatha. Each predicate symbol P of arity n is interpreted as an n-ary relation over the domain, *i.e.*

$$F(P) \subseteq \underbrace{D \times \ldots \times D}_{n \text{ times}}$$

For example $F(Corpse)$ is some subset of D, which we can specify as the set of corpses within the domain. Another example is $F(HasMotiveToMurder)$ which is some subset of $D \times D$, which we can specify as being the set of pairs of people in the domain where the first person in the pair has a motive to murder the second. Each function symbol f of arity m is interpreted as an m-ary function over the domain, *i.e.*

$$F(f) \subseteq \underbrace{D \times \ldots \times D}_{n \text{ times}} \to D$$

For example $F(KillerOf)$ is some function $D \to D$, which we can specify as being the function which maps a person to his or her killer.

Note that there can be multiple models for a given vocabulary with differing domains and interpretation functions.

Given a particular vocabulary, a model for that vocabulary and a formula over that vocabulary, we are interested in making some kind of evaluation of the formula (description) with respect to the model (situation). So far we have only seen how vocabulary elements are to be interpreted. In order to interpret the variables of our first-order formulae, we introduce an assignment function α which maps from the set of variables to the model domain, *i.e.* $\alpha(x) \in D$ for variable x and domain D. This function, by mapping variables to elements in the domain, can be thought to assign contextual information. We then are able to talk about the 'satisfaction' of a formula in the model with respect to a particular assignment function. Before we can formally define the notion of satisfaction, we give two further definitions.

Let $M \equiv (D,F)$ be a model and let α be an assignment function which maps variables to elements in D. Let τ be a term. We denote the 'interpretation of τ with respect to F and α' as $I_F^\alpha(\tau)$ and define it as follows.

$$I_F^\alpha(\tau) \equiv \begin{array}{l} F(\tau) \text{ if } \tau \text{ is a constant or function} \\ \alpha(\tau) \text{ if } \tau \text{ is a variable} \end{array}$$

Now suppose β is another assignment function which maps variables to elements in D. Let $x, y, z, \ldots$ be the infinite set of variables of our first-order language. Suppose $\beta(x) \neq \alpha(x)$. Suppose however that for each and every variable distinct from x, $\beta(y) = \alpha(y)$ and $\beta(z) = \alpha(z)$, *etc.* Then we say β is an x-variant of α. Variant assignments allow us to try out new values for a given variable (say, x) while keeping the values assigned to all other variables the same. We now define the relation $M, \alpha \vDash \varphi$ (which can be read 'formula φ is satisfied in M with respect to assignment α') as follows.

$M, \alpha \vDash P(\tau_1, \ldots, \tau_n)$	iff	$(I_F^\alpha(\tau_1), \ldots, I_F^\alpha(\tau_n)) \in F(P)$
$M, \alpha \vDash \neg\varphi$	iff	not $M, \alpha \vDash \varphi$
$M, \alpha \vDash \varphi \wedge \psi$	iff	$M, \alpha \vDash \varphi$ and $M, \alpha \vDash \psi$
$M, \alpha \vDash \varphi \vee \psi$	iff	$M, \alpha \vDash \varphi$ or $M, \alpha \vDash \psi$
$M, \alpha \vDash \varphi \Rightarrow \psi$	iff	not $M, \alpha \vDash \varphi$ or $M, \alpha \vDash \psi$
$M, \alpha \vDash \forall x.\, \varphi$	iff	$M, \beta \vDash \varphi$ for all x-variants β of α
$M, \alpha \vDash \exists x.\, \varphi$	iff	$M, \beta \vDash \varphi$ for some x-variant β of α
$M, \alpha \vDash \tau_1 = \tau_2$	iff	$I_F^\alpha(\tau_1) = I_F^\alpha(\tau_2)$

The symbol $\vDash$ is usually referred to as the satisfaction relation. Note that if term τ is of the form $f(\tau 1, \ldots, \tau_m)$ for a function f of m terms, then $I_F^\alpha(\tau)$ is defined to be $F(f)(I_F^\alpha(\tau_1), \ldots, I_F^\alpha(\tau_m))$.

Since a vocabulary may have many possible models, a formula over that vocabulary may be satisfied in one model and not in another. We write the set of all possible models over a given vocabulary as $\mathcal{M}$. We say a formula is satisfiable if it is satisfied in at least one model of $\mathcal{M}$ (with respect to a given assignment function) and unsatisfiable otherwise. This notion can be extended to finite sets of formulae. A finite set of formulae $\{\varphi_1, \ldots, \varphi_n\}$ is satisfiable if $\varphi_1 \wedge \ldots \wedge \varphi_n$

is satisfiable. Similarly $\{\varphi_1,\ldots,\varphi_n\}$ is unsatisfiable if $\varphi_1 \wedge \ldots \wedge \varphi_n$ is unsatisfiable. Essentially, satisfiable formulae can be thought of as describing conceivable, possible, or realisable situations. Unsatisfiable formulae describe inconceivable, impossible situations. A simple example of an unsatisfable formula is $\varphi \wedge \neg \varphi$.

We say a formula is valid if it is satisfied in all models of $\mathcal{M}$ given any variable assignment, and invalid otherwise. The notation $\models \varphi$ is used to indicate that a formula φ is valid. A simple example of a valid formula is $\varphi \vee \neg \varphi$. In logic, validity is often considered in terms of logical arguments or inferences. We say that an argument with premises $\varphi_1,\ldots,\varphi_n$ and conclusion ψ is valid if and only if whenever all the premises are satisfied in some model, using some variable assignment, then the conclusion is satisfied in that same model using the same variable assignment. We use the notation $\varphi_1,\ldots,\varphi_n \models \psi$ to indicate that the argument with premises $\varphi_1,\ldots,\varphi_n$ and conclusion ψ is valid. We also say that ψ is a logical consequence of $\varphi_1,\ldots,\varphi_n$, or that $\varphi_1,\ldots,\varphi_n$ logically entails ψ. (Here the $\models$ symbol refers to a semantic entailment relation rather than a satisfaction relation; the overloading of the symbol $\models$ is traditional.) Importantly, every valid argument $\varphi_1,\ldots,\varphi_n \models \psi$ corresponds to the valid formula $\models \varphi_1 \wedge \ldots \wedge \varphi_n \Rightarrow \psi$. Moreover, two formulae φ and ψ are said to be logically equivalent if and only if both $\varphi \models \psi$ and $\psi \models \varphi$.

We want to describe a DRS language translation to a first-order language built over the same vocabulary. In order for such a translation to make sense, we need to interpret both languages in the same way over the same model. Hence we need to define an interpretation for DRSs that (ideally) makes use of the same semantic machinery as first-order languages.

Suppose we have a model $M \equiv (D,F)$ for a given DRS vocabulary, then we define an embedding in M as a function which maps from the set of discourse referents to D. Intuitively, the embedding assigns context. (The function is called an embedding since it can be thought of as embedding the DRS within M. Since discourse referents are DRT terminology for variables, an embedding is simply an assignment function.) We distinguish between the semantics of conditions and the semantics of DRSs. The semantics of DRS conditions is treated statically. The relation $M,\alpha \models C$ has the meaning that condition C is satisfied in a model M with respect to embedding α. The semantics of DRSs is treated dynamically. Since DRSs are designed to capture contextual change, we reflect this in the semantics by introducing two embeddings: the input and output embedding. We use the notation $\alpha[x_1,\ldots,x_n]\beta$ to indicate that β differs from α only in the values it assigns to the discourse referents $x_1,\ldots,x_n$; this is just a n-places version of a variant assignment described earlier. We define $I_F^{\alpha}(\tau)$ – the interpretation of term τ with respect to F and α – as $F(\tau)$ if τ is a constant or function, and $\alpha(\tau)$ if τ is a discourse referent. The relation $M,\alpha,\beta \models B$ has the meaning that DRS B is satisfied in M with respect to the embeddings α and β. The satisfaction relation is defined as follows.

$$
\begin{array}{lll}
M,\alpha \models P(\tau_1,\ldots,\tau_n) & \text{iff} & (I_F^{\alpha}(\tau_1),\ldots,I_F^{\alpha}(\tau_n)) \in F(P) \\
M,\alpha \models \tau_1 = \tau_2 & \text{iff} & I_F^{\alpha}(\tau_1) = I_F^{\alpha}(\tau_2) \\
M,\alpha \models \neg B & \text{iff} & \text{for all } \beta, \text{ not } M,\alpha,\beta \models B \\
M,\alpha \models B_1 \vee B_2 & \text{iff} & \text{there is a } \beta \text{ such that } M,\alpha,\beta \models B_1 \text{ or } M,\alpha,\beta \models B_2 \\
M,\alpha \models B_1 \Rightarrow B_2 & \text{iff} & \text{for all } \beta \text{ such that } M,\alpha,\beta \models B_1 \text{ there is a } \delta \\
 & & \text{such that } M,\beta,\delta \models B_2
\end{array}
$$

Finally,

$$
M,\alpha,\beta \models \begin{array}{|c|}\hline x_1 \ldots x_n \\ \hline C_1 \\ \vdots \\ C_m \\ \hline \end{array} \quad \text{iff } \alpha[x_1,\ldots,x_n]\beta \text{ and } M,\beta \models C_1 \text{ and } \ldots \text{ and } M,\beta \models C_m
$$

The first two relations are self-explanatory. The third relation says that condition $\neg B$ is satisfied in M with respect to context α when it is not possible to update DRS B in context α to a new context β such that B is satisfied in β. The relation for disjunctive conditions says that $B_1 \vee B_2$ is satisfied in M with respect to context α if we can update either DRS B_1 or DRS B_2 in context α to a new context β such that respectively, B_1 or $B_2$is satisfied in β. The relation for implicational conditions says that $B_1 \Rightarrow B_2$ is satisfied in M with respect to context α if for any context β (where β is such that DRS B_1 in context α can be updated to β and $M,\alpha,\beta \models B_1$) there is a context δ such that DRS B_2 in β can be updated to δ and $M,\alpha,\delta \models B_2$. The final relation says that an updated DRS (with discourse referents $x_1,\ldots,x_n$ and conditions $C_1,\ldots,C_m$) is satisfied in M with respect to input context α and output context β if β differs from α only in the values it assigns to $x_1,\ldots,x_n$ and if each condition $C_1,\ldots,C_m$ is satisfied in M with respect to β.

We say a DRS B is dynamically satisfied in a model M with respect to an embedding α if and only if there is an embedding β such that $M,\alpha,\beta \models B$. If B is dynamically satisfied in M with respect to α, we write $M,\alpha, \models B$.

We now follow (Blackburn and Bos 1999) and show how to translate DRSs into formulae of (full) FOL with equality. We define a translation function *fo* which maps a DRS built over some vocabulary into formulae of the first-order language built over that same vocabulary. The translation function *fo* is satisfaction preserving, namely, $M,\alpha, \models B$ iff $M,\alpha, \models fo(B)$. We won't prove this here; instead we refer the interested reader to (Blackburn and Bos 1999).

A general DRS is mapped to the following FOL formula.

$$
fo\left(\begin{array}{|c|}\hline x_1 \ldots x_n \\ \hline C_1 \\ \vdots \\ C_m \\ \hline \end{array}\right) \equiv \exists x_1,\ldots,x_n.\,(fo(C_1) \wedge \ldots \wedge fo(C_m))
$$

If there are no conditions in the condition set, then the translation is $\exists x_1,\ldots,x_n \top$ where $\top$ can be thought of as an atomic formula which is always true in any given model with respect to any given assignment. If the DRS has an empty universe, *i.e.* it has no discourse referents, then the translation is $fo(C_1) \wedge \ldots \wedge fo(C_m)$. Hence the translation of an empty DRS is $\top$.

Primitive conditions are simply mapped to themselves.

$$fo(P(\tau_1,\ldots,\tau_n)) \equiv P(\tau_1,\ldots,\tau_n)$$
$$fo(\tau_1 = \tau_2) \equiv \tau_1 = \tau_2$$

Complex conditions involving the connectives $\neg$ and $\vee$ are mapped such that the translation function is pushed in over the connective.

$$fo(\neg B) \equiv \neg fo(B)$$
$$fo(B_1 \vee B_2) \equiv fo(B_1) \vee fo(B_2)$$

Complex conditions involving $\Rightarrow$ are translated as follows.

$$fo\left(\begin{array}{|c|}\hline x_1 \ldots x_n \\ \hline C_1 \\ \vdots \\ C_m \\ \hline\end{array} \Rightarrow B\right) \equiv \forall x_1,\ldots,x_n.\,(fo(C_1) \wedge \ldots \wedge fo(C_m) \Rightarrow fo(B))$$

If there are no conditions in the condition set of the antecedent, then the translation is $\forall x_1,\ldots,x_n\,(\top \Rightarrow fo(B))$ which is logically equivalent to $\forall x_1,\ldots,x_n\,.\,\top \Rightarrow fo(B))$. If the DRS has an empty universe, then the translation is $fo(C_1) \wedge \ldots \wedge fo(C_m) \Rightarrow fo(B)$. Hence if the antecedent DRS is empty then we obtain the translation $\top \Rightarrow fo(B))$ which is logically equivalent to $fo(B)$.

3.6.2 DRS Construction in PENG

As outlined in (Schwitter 2004a; Schwitter and Tilbrook 2004b), a DRS is represented in PENG as the Prolog term `drs(U,Con)` consisting of a list `U` of discourse referents `[I1,I2,…,In]` and a list `Con` of conditions `[C1,C2,…,Cn]`. The primitive conditions of a PENG DRS can only be formed using the predicate symbols `obj`, `struc`, `named`, `pred`, `evtl`, `prop` and `role`. For example, a DRS condition such as *Gun*(*I*) is represented in PENG as a Prolog list of two conditions `[obj([gun],I),struc(I,atomic)]`. (Note this list is the value for the `con` attribute in the lexical entry for the noun *gun*.) Here the discourse referent `I` denotes an object with an atomic structure which falls under the concept `gun`. By describing concepts using a limited set of metadata predicates, PENG avoids having to introduce a predicate for every concept. It thus avoids having to perform higher-order quantification by quantifying over predicates. Ultimately, simpler DRSs are constructed, which in turn creates less work for the inference tools.

DRS conditions derived from nouns describe objects. Further PENG examples include: `[obj([police],I),struc(I,group)]` and `[obj([blood],I),struc(I,mass)]`. Lexical entries for proper nouns contain conditions describing a name, for example `[named([agatha],I),struc(I,atomic)]`. A name is assigned a structure.

DRS conditions derived from verbs describe eventualities. An eventuality can be classified as either an event or state. Each verb introduces an additional discourse referent representing that event or state. Event verbs denote a change in time whereas state verbs express static properties. The transitive verb *suspects* is represented `[pred(E,[suspects],I1,I2),evtl(E,event)]`. Here *suspects* is in the context of *the detective suspects Agatha*. A transitive verb *has* is represented in PENG as `[pred(S,[has],I1,I2),evtl(S,state)]`. Here *has* is in the context of *the detective has a trenchcoat*.

Lexical entries for adjectives contain a single DRS condition, *e.g.* `[prop([dangerous],I)]`. DRS conditions for adverbs have an additional condition which specifies their role, *e.g.* `[prop(M,[allegedly],I),role(M,manner)]`. For further discussion on the flattened notation for primitive DRS conditions see (Schwertel 2005).

We saw in Section 3.6 how the standard DRS construction algorithm builds a DRS from a sentence and its corresponding parse tree. The algorithm moves around the tree top-down, left-to-right, gathering the semantic information of the various sentence constituents at each node and placing this information into a DRS. The DRS is then used as the context for processing the second sentence, and so on. The construction algorithm employed by PENG takes a slightly different approach: it uses the concept of DRS threading (Blackburn and Bos 1999). Here imagine an algorithm which slides a DRS from node to node top-down, left-to-right. When the DRS slides over a node – or in other words, when the node threads through the DRS – the node places the semantic information held at that location within the DRS. We can think of an incoming and outgoing DRS existing at every node. The difference between the two DRSs is exactly the information that is contributed at each node.

The relationship between an incoming and outgoing DRS is modelled by a Prolog difference list `DrsIn-DrsOut`. (Recall from Section 3.4.1 that a difference list is a pair of lists – the first the input list, the second the output list – whereby the information of interest is the difference between the two lists.) Hence the incoming DRS is represented by the input list `DrsIn`, and the outgoing DRS is represented by the output list `DrsOut`. Consider, for example, the parser's processing of the noun *butler*. Recall from Section 3.5.1 that we have the following simplified lexical entry.

```
lexicon([lex:[butler],synon:[pantryman]],
        [cat:cn,
         arg:[ind:I,type:person,agr:[per:third,num:sg,gend:_],
              case:_],
         con:[obj([butler],I),struc(I,atomic)]],base).
```

From the same section we have the following simplified production rule. Note that here we have included the `drs` attribute for the first time.

```
n0([cat:cn,
    arg:[ind:I|Rest],
    drs:[drs(U1,C1)|D]-[drs([I|U1],[C3,C2|C1])|D]])
    -->
   {lexicon([lex:Noun],
            [cat:cn,
             arg:[ind:I|Rest],
             con:[C3,C2]]},
   Noun.
```

During processing, the chart parser unifies the `con` attribute-value pair of the lexical entry and the production rule. Hence `obj([butler],I)` and `struc(I,atomic)` are unified with the variables `C3` and `C2`. These variables are then added in the outgoing DRS list to the conditions `C1` of the incoming DRS. The discourse referent `I` is added in the outgoing DRS to the universe `U1` of the incoming DRS. Hence we have an ingoing DRS with a list of discourse referents `U1` and a list of conditions `C1`; and an outgoing DRS such that the referent `I` heads the list (with tail `U1`), and conditions `C3` and `C2` head the list (with tail `C1`).

Consider a similar example: the processing of the verb *argue*. We have the following simplified lexical entry.

```
lexicon([lex:[argue],synon:[]],
        [cat:tv,
         arg:[ind:I1|Rest],arg:[ind:I2|Rest],
         con:[pred(E,[argue],I1,I2),evtl(E,event)]],base).
```

We have the following simplified production rule (Schwitter 2004a).

```
v0([cat:tv,
    arg:[ind:I1|Rest],arg:[ind:I2|Rest],
    drs:[drs(U1,C1)|D]-[drs([E|U1],[C3,C2|C1])|D]])
    -->
   {lexicon([lex:Verb],
            [cat:tv,
             arg:[ind:I1|R],arg:[ind:I2|Rest],
             evtl:E,
             con:[C3,C2]]},
   Verb.
```

During processing, the chart parser unifies the `con` attribute-value pair of the lexical entry and the production rule. Hence `pred(E,[argue],I1,I2)` and `evtl(E,event)` are unified with the variables `C3` and `C2`. The variables are then added in the outgoing DRS list to the conditions `C1` of the incoming DRS. The variable `E` representing the eventuality is added to the list of discourse referents `U1` of the outgoing DRS.

The chart parser's processing of determiners is of particular interest. Here's the simplified production rule for the determiner *the* (Schwitter and Tilbrook 2004b). (The determiner *the* is a definite article; its definiteness is specified by the `spec:def` attribute-value pair.)

```
d0([cat:det,
    arg:[agr:G|Rest],
    spec:def,
    drs:D1-D3,
    res:[drs([],[])|D1]-D2,
    sco:D2-D3])
    -->
   {lexicon([lex:Determiner],
            [cat:det,
             arg:[agr:G|Rest],
             spec:def])},
   Determiner.
```

Within a sentence, each determiner includes two arguments: a restrictor and a scope, represented by the attribute `res` and `sco` respectively. The restrictor consists of the remaining noun phrase. The scope consists of the rest of the sentence outside the noun phrase. Since a

definite noun phrase may be used anaphorically, the chart parser builds a store DRS while processing the phrase. This initially empty DRS `drs([],[])` is placed in front of the restrictor's incoming DRS `D1`. As the noun phrase is processed, the store collects all the discourse referents and conditions for the noun phrase. Once the entire noun phrase has been processed, the store can be accessed by the anaphora resolution algorithm. After anaphora resolution, the restrictor's outgoing DRS `D2` will contain the resolved DRS conditions that are then passed to the scope's incoming DRS. After processing the verb phrase, the scope's outgoing DRS `D3` will contain the semantic information for the entire sentence. We can see this process taking place if we unify the above rule with the simplified version of the `n3` rule presented earlier in Section 3.5.1.

```
n3([...,
    arg:[ind:I|Rest],
    drs:D,
    sco:S,
    ...])
    -->
    d0([...,
        drs:D,
        res:R1-R3,
        sco:S,
        ...]),
    n2([cat:cn,
        arg:[ind:I|Rest],
        drs:R1-R2,
        ...]),
    {anaphora_resolution(n3,cn,I,R2,R3,...)}.
```

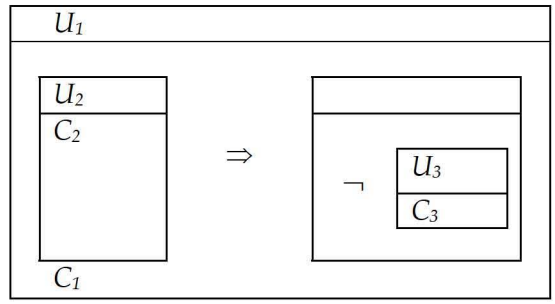
We see that the ingoing restrictor `R1` (having been unified with `[drs([],[])|D1]`) is passed to `n2` where the remaining semantic information of the noun phrase is accumulated. Once the entire `n3` noun phrase has been processed, the outgoing DRS `R2` of `n2` will have the form `[B|D1]` where `B` is the DRS representing the noun phrase, and `D1` is the ingoing DRS of the determiner. The anaphora resolution algorithm of `n3` checks whether `B` is accessible from `D1`. After anaphora resolution, the restrictor's outgoing DRS `R3` (having been unified with `D2`) will contain the resolved DRS conditions which are then passed to the scope's incoming DRS. (Note that the `sco:S` attribute-value pair of the `n3` rule unifies with the `sco:D2-D3` pair of the determiner's lexical entry.)

We next consider the processing of the indefinite determiner *no*. It is used in the context *no detective smokes a pipe*. Here's a production rule for such a determiner (Schwitter and Tilbrook 2004b).

```
d0([cat:det,
    arg:[agr:G|Rest],
    spec:no,
    drs:D1-[drs(U1,[drs(U2,C2)->drs([],[~drs(U3,C3)])|C1])|D3],
    res:[drs([],[])|D1]-D2,
    sco:[drs([],[])|D2]-[drs(U3,C3),drs(U2,C2),drs(U1,C1)|D3])
    -->
   {lexicon([lex:Determiner],
            [cat:det,
             arg:[agr:G|Rest],
```

```
                spec:no])},
    Determiner.
```

During processing, the chart parser places a store DRS `drs([],[])` in front of the restrictor's incoming DRS `D1`. This initially empty DRS collects all the discourse referents and conditions for the noun phrase. The scope then takes the restrictor's outgoing DRS `D2` and again places an empty DRS before it. This store DRS collects all the discourse referents and conditions outside the noun phrase. The DRS for the restrictor and the scope are then embedded into a complex condition – consisting of an implication and a negation – representing the meaning of the negative determiner. The complex condition `drs(U1,[drs(U2,C2)->drs([],[~drs(U3,C3)])|C1])` can be represented in our graphical notation as follows.



3.7 Nonfirstorderisable Sentences

Before we discuss the reasoning services of the PENG system, it's worth making some brief comments on the restriction first-order logic places on the PENG grammar. We should state right away that the word 'nonfirstorderisable' is used in the Philosophical Logic literature; it's usually credited to George Boolos (Boolos 1984). Nonfirstorderisable sentences refer to English sentences which cannot be represented in FOL. Examples – taken from (McKay 2006) – include the following.

They are classmates
They are meeting together
They are surrounding a building
They lifted a piano
They admire only one another
The rocks rained down
The seashells are scattered
The mechanics repaired the car
The musicians will perform the symphony
The chairs form a circle

These sentences are nonfirstorderisable because any predicate symbols introduced to represent the notions *are classmates, are meeting together, are surrounding a building etc.* are not 'distributive'. A predicate symbol *P* is said to be distributive if, whenever some things have the property described by *P*, then each thing has that same property. In FOL every predicate symbol is distributive. For example, whenever some people are groundskeepers, each one of them is a groundskeeper. However, our nonfirstorderisable sentences can be true of some

classmates, some people who are meeting together and some people who are surrounding a building *etc.*, without being true of any one of the individuals described. For example two students might be classmates, but one student by herself cannot be a classmate. She must be a classmate of *someone*.

Further examples of nonfirstorderisable sentences include the following.

There are fewer than four in number

They are a minority

They are of just one gender

They are odd in number

These sentences are nonfirstorderisable because any predicate symbols introduced to represent the notions *are fewer than four*, *are a minority*, *are of just one gender* etc. are not 'cumulative'. A predicate symbol P is said to be cumulative if, whenever some set of things X have the property described by P, and some set of things Y have the property described by P, then X and Y together have that same property. For example, if Frank and Bob are groundskeepers, and Maureen is a groundskeeper, then Bob, Frank and Maureen are (cumulatively) groundskeepers. In contrast, if X and Y are both sets of less than four objects, then X and Y together might (non-cumulatively) have four or more objects. In FOL, every predicate is cumulative. Many predicates which are not distributive are also not cumulative. However, there are a number of predicates that are distributive but are not cumulative.

As can be seen in the previous examples, the majority of nonfirstorderisable sentences feature plural quantification. Contrast the sentence *there is an apple on the table*, which features singular quantification with the quantification in *there are some apples on the table*, which is plural. Normally we paraphrase such a sentence in FOL, *i.e.*

$$\exists x,y.\ (AppleOnTable(x) \wedge AppleOnTable(y) \wedge \neg(x = y))$$

However such a representation is inaccurate since we don't know how many apples are on the table; we only know that there are 'some'. Moreover, the predicate symbol *AppleOnTable* is cumbersome. As discussed in (Rayo 2002), the plural counterparts to $\forall x$ and $\exists y$ are $\forall xx$ and $\exists yy$, which can be read 'for any objects xx' and 'there are some objects yy'. (Here xx and yy can be thought of as a plural variable.) In order to logically represent nonfirstorderisable sentences, we can extend a first-order language with plural quantifiers, along with formulae of the form $u \prec xx$. Such formulae are given the semantics 'u is one of the objects xx'. The language extension is usually referred to as a Plural First-Order (PFO) language. As given in (Linnebo 2005) the PFO representation of the nonfirstorderisable sentence *there are some apples on the table* is

$$\exists xx \forall y.(y \prec xx \Rightarrow Apple(y) \wedge OnTable(y))$$

Another example is the Geach-Kaplan sentence *some critics admire only one another* which resists a FOL paraphrase. This sentence can be given the following PFO representation.

$$\exists xx \forall y,z.(y \prec xx \Rightarrow Critic(y) \wedge (Admires(y,z) \Rightarrow z \prec xx \wedge \neg(y = z)))$$

Because an English sentence can be formulated in PENG only as long as the sentence can be represented in FOL, the set of possible PENG sentences is a subset of the set of firstorderisable sentences. There are restrictions on the grammar that shrink this subset further; we refer the reader to Appendix A.

4. Reasoning

In this section we examine the reasoning services of the PENG system. Having translated its DRSs to FOL using the translation described in Section 3.6.1, PENG relies on third-party reasoning services – the theorem prover Otter and the model builders Mace4 and Satchmo – for consistency and informativity checking, and for question/answering. Before we describe these tools in Sections 4.2.1-4.2.3 respectively, we look at the tableau and resolution proof methods for FOL and briefly discuss the advantages of model building *vs.* theorem proving. Much of Section 4.1 is derived from (Blackburn and Bos 2005).

4.1 Inference Procedures for First-Order Logic

There are three inference tasks fundamental to the field of computational semantics: query; consistency checking; and informativity checking. Given a particular vocabulary, a model M for that vocabulary and a first-order formulae φ over that vocabulary, a query task asks whether φ is satisfied in M. As long as the models are finite, the querying task can be straightforwardly handled by a first-order model checker. The building of such a checker – implemented in Prolog – is described in (Blackburn and Bos 2005).

Given a particular vocabulary, the set of all possible models $\mathcal{M}$ for that vocabulary and a first-order formulae φ over that vocabulary, a consistency check asks whether φ is consistent (meaning that it is satisfied in at least one model of $\mathcal{M}$) or inconsistent (meaning that φ is satisfied in no model of $\mathcal{M}$). We mentioned previously that a formula is said to be satisfiable if it is satisfied in at least one model, hence consistency is usually identified with satisfiability, and inconsistency with unsatisfiability. Consistency checking for first-order formulae is computationally undecidable, meaning that there is no algorithm capable of solving this problem for all input formulae φ. Not only must a satisfying model be found amongst the vast number of possible models, but that satisfying model must be finite. However, some formulae only have satisfying models which are infinite in size.

Given a particular vocabulary, the set of all possible models $\mathcal{M}$ for that vocabulary and a first-order formulae φ over that vocabulary, an informativity check asks whether φ is informative (meaning that it is not satisfied in at least one model of $\mathcal{M}$) or uninformative (meaning that φ is satisfied in all models of $\mathcal{M}$). Since a formula is invalid if there is at least one model in which it is not satisfied, and is valid if it is satisfied in all models, we usually identify informativity with invalidity and uninformativity with validity. Valid formulae can be seen to be uninformative since they don't tell us anything new about a particular model. For example *HasSibling*(*agatha*) is uninformative with respect to *HasBrother*(*agatha*). Such formulae should not be entirely disregarded; often it is appropriate to rephrase the same information. Informativity checking for first-order formula is also undecidable.

Derived from their definitions, consistency and informativity are related as follows.

1. φ is consistent if and only if $\neg\varphi$ is informative.
2. φ is inconsistent if and only if $\neg\varphi$ is uninformative.
3. φ is informative if and only if $\neg\varphi$ is consistent.
4. φ is uninformative if and only if $\neg\varphi$ is inconsistent.

For example suppose φ is consistent. This means it is satisfied in at least one model, which is the same as saying that there is at least one model in which $\neg\varphi$ is not satisfied. Hence $\neg\varphi$ is informative. Because of these inter-relations, both consistency and informativity checks can therefore be reformulated in terms of validity. We say ψ is uninformative with respect to premises $\varphi_1,\ldots,\varphi_n$ if and only if the formula $\varphi_1 \wedge\ldots\wedge \varphi_n \Rightarrow \psi$ is valid, and ψ is inconsistent with respect to $\varphi_1,\ldots,\varphi_n$ if and only if the formula $\varphi_1 \wedge\ldots\wedge \varphi_n \Rightarrow \neg\psi$ is valid.

An (undecidable) theorem prover can be used to determine whether a first-order formula is valid. These programs usually implement tableau or resolution-based proof methods.

4.1.1 The Tableau Proof Method

The following description of the tableau proof method follows (Blackburn and Bos 2005). Given a formula φ, the tableau proof method checks its validity by proving that $\neg\varphi$ is unsatisfiable. Moreover, the method checks the validity of an inference with premises $\varphi_1,\ldots,\varphi_n$ and conclusion ψ by proving the set $\{\varphi_1,\ldots,\varphi_n, \neg\psi\}$ unsatisfiable. A tree is constructed – called a tableau – such that formulae in leaf nodes of the same branch are conjuncted, whereas different branches are disjuncted. Applicable rules of a tableau calculus are applied (in any order) top-down to each node. These rules specify how each logical connective is to be broken down. Complex formulae are eventually broken into atomic formulae (or their negation) until the tree becomes rule-saturated. At this point the tree can no longer be expanded. A branch containing an opposite pair of literals is called closed. A literal is simply an atomic formula (which may contain free variables) or the negation of an atomic formula. If all branches of the tableau are closed, then we can be said to have found a tableau proof for the set of formulae, meaning that the set of formulae is unsatisfiable.

We first consider the tableau proof method for a formula of propositional logic. Propositional logic is a particular quantifier-free fragment of FOL. Although the atomic formulae of propositional logic may contain free variables, nothing is lost by replacing them with simpler, variable-free symbols such as p, q and r. This is because no free variables can be bound – there are no quantifiers – and hence the internal structure of an atomic formula is unimportant. The symbols p, q and r are usually called propositions; they may also be referred to as literals.

Consider the validity check for the propositional logic formula $(p \vee \neg q) \wedge q \Rightarrow p$. We construct a tableau/tree for the set $\{(p \vee \neg q) \wedge q \neg p\}$. Initially we have

$$(p \vee \neg q) \wedge q$$
$$|$$
$$\neg p$$

The conjunctive tableau rule says that if a branch of the tableau contains a formula $p \wedge q$then add to its leaf the chain of two nodes containing the literals p and q. We formalise this rule as

$$\frac{p \wedge q}{\begin{array}{c} p \\ q \end{array}}$$

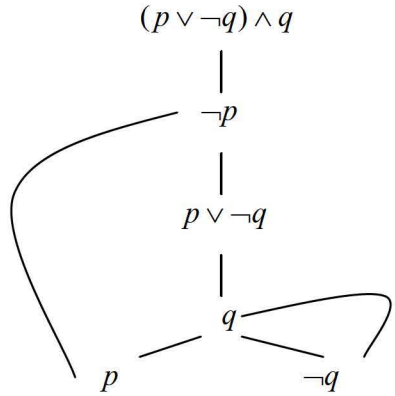
The rule is read from top to bottom; the top being the input to the rule, the bottom being the output. Hence after application of this rule we have the following tableau.

$$\begin{array}{c} (p \vee \neg q) \wedge q \\ | \\ \neg p \\ | \\ p \vee \neg q \\ | \\ q \end{array}$$

The disjunctive tableau rule says that if a branch of the tableau contains a formula $p \vee q$ then create two sibling children to the leaf of the branch containing p and q respectively. We write this formally as

$$\frac{p \vee q}{p \mid q}$$

Applying this rule gives us



Since both left and right branches of the tableau are closed, we can deduce that the set $\{(p \vee \neg q) \wedge q \neg p\}$ is unsatisfiable. Hence our original formula $(p \vee \neg q) \wedge q \Rightarrow p$ is valid.

We can extend the tableau method for formulae of FOL by incorporating two rules for universal and existential quantifiers. The universal rule is given below.

$$\frac{\forall x.\, \gamma(x)}{\gamma(x')}$$

Suppose we chose x' to be an arbitrary term. Then x' can be selected such that the tableau never closes. A solution is to delay the choice of the term until the consequent of a rule application allows us to close at least one branch of the tableau. An application of the universal rule generates $\chi(x')$ where x' is chosen such that it does not occur anywhere else in the tableau. Later x' is substituted – throughout the entire tableau – with the most general unifier such that at least one branch is closed. Multiple applications of the universal rule may be applied to the same node of the tableau. As an example, the set $\{\neg P(a) \vee \neg P(b), \forall x.\ P(x)\}$ can only be proved unsatisfiable if both $P(x')$ and $P(x'')$ are generated from $\forall x.\ P(x)$ and then x' is substituted with a and x'' with b.

The existential rule is given below.

$$\frac{\exists x.\ \chi(x)}{\chi(f(x_1,\ldots x_n))}$$

This rule performs skolemisation, meaning that every existentially quantified variable x is replaced with $f(x_1,\ldots,x_n)$ where f is a new function symbol and $x_1,\ldots,x_n$ denotes the free variables of χ that are universally quantified with $\exists x$ in their scope. For example the formula $\forall x \exists y \forall z.\ P(x,y,z)$ is skolemised to $\forall x \forall z.\ P(x, f(x),z)$. The skolem term $f(x)$ contains x but not z, since the quantifier $\exists y$ is in the scope of $\forall x$ but not $\forall z$.

The tableau proof method for first-order formulae constructs a tableau in a similar fashion to the propositional case, with the additional assumption that all free-variables are universally quantified. For example if φ is the consequent formula of a rule application, and $x_1,\ldots,x_n$ are the free variables of φ, then $\forall x_1,\ldots,x_n.\varphi$ is the formula represented by the tableau at that node.

As discussed in (Fitting and Mendelsohn 1998), a proof procedure describes how tableau rules should be applied in order to close the branches of a tableau. A tableau calculus is said to be complete if we can construct a tableau proof for every given unsatisfiable set of formulae. However, even if a calculus is complete, not every possible choice of a rule application will lead to a proof for an unsatisfiable set. A general solution is to search the tableau space for a given set of formulae until a closed tableau is found. The tableau space consists of all tableau generated by the different combinations of rule applications. There are various ways of searching the tree structure of the tableau space. Some techniques search breadth-first rather than depth-first. Some search methods use iterative deepening, whereby each branch of the tableau space is visited up to a certain depth, the depth is then increased and further search is undertaken. Some other techniques disallow the generation of particular tableau (based on their structure) within the tableau space.

An overview of tableau proof methods can be found in (D'Agostino, Gabbay et al. 1999). Theorem provers that implement tableau-based proof methods for FOL formulae include LeanTAP (Beckert and Posegga 1997) and 3TAP (Beckert, Hähnle et al. 1996).

4.1.2 The Resolution Proof Method

The following description of the resolution proof method follows from (Blackburn and Bos 2005). We begin by looking at resolution from a propositional perspective.

Before performing resolution we first need to convert a formula of propositional logic into Conjunctive Normal Form (CNF). A formula is in CNF if and only if it is a conjunction of clauses. (By clause we mean a disjunction of literals.) For example the formula $(p \vee q) \wedge (r \vee \neg p \vee s) \wedge (q \vee \neg s)$ is in CNF. Usually a clause is given a list representation, *e.g.* $p \vee q$ is written as $[p,q]$. Furthermore, the connective $\wedge$ is given a list-of-lists representation. Hence our previous example can be written as $[[p,q],[r,\neg p,s],[q,\neg s]]$. A list-of-lists representation may also contain the empty clause $[\ ]$ which is always false. (Essentially, $[\ ]$ is logically equivalent to $\bot$, which can be thought of as an atomic formula which is always false in any given model with respect to any given assignment.) An important point is that if a formula in CNF is true, then all of its clauses must be true. Hence if a formula contains an empty clause it cannot be true.

In order to transform a formula of propositional logic into CNF we first convert it to Negated Normal Form (NNF). A formula is in NNF if and only if the formula is built from literals using $\wedge$ and $\vee$ as the only binary connectives. To convert a formula into NNF we perform the following rewrites.
(Rewrite 1) $\neg(\varphi \wedge \psi)$ as $\neg\varphi \wedge \neg\psi$.
(Rewrite 2) $\neg(\varphi \wedge \psi)$ as $\neg\varphi \wedge \neg\psi$.
(Rewrite 3) $\neg(\varphi \Rightarrow \neg\psi)$ as $\varphi \wedge \neg\psi$.
(Rewrite 4) $\varphi \Rightarrow \psi$ as $\neg\varphi \vee \psi$.
(Rewrite 5) $\neg\neg\varphi$ as φ.

Once a formula is in NNF we can then apply the following distributive and associative rewrites. The associative rewrites allow brackets to be moved around so that the distributive rewrites may be applied. The distributive rewrites drive $\vee$ deeper into the formula and 'lift out' $\wedge$, converting the formula into CNF.
(Rewrite 6) $\theta \vee (\varphi \wedge \psi)$ as $(\theta \vee \varphi) \wedge (\theta \vee \psi)$.
(Rewrite 7) $(\varphi \wedge \psi) \vee \theta$ as $(\varphi \vee \theta) \wedge (\psi \vee \theta)$.
(Rewrite 8) $(\varphi \wedge \psi) \wedge \theta$ as $\theta \wedge (\varphi \wedge \psi)$.
(Rewrite 9) $(\varphi \vee \psi) \vee \theta$ as $\theta \vee (\varphi \vee \psi)$.

For example the formula $(\neg p \Rightarrow q) \Rightarrow (\neg r \Rightarrow s)$ can be converted to the following NNF $(\neg p \wedge \neg q) \vee (r \vee s)$ using rewrites 4, 3 and 4, and then 5. Rewrite 7 can then be applied to obtain the CNF $(\neg p \vee (r \vee s)) \wedge (\neg q \vee (r \vee s))$. This can then be written in list-of-lists notation as $[[\neg p,r,s],[\neg q,r,s]]$ and is termed a clause set. It is worth noting that multiple applications of the two distribution rewrites 6 and 7 can cause exponential blow-up relative to the size of the input formula. There are workarounds which usually involve introducing new variables for sub-formulas; see (Leitch 1997) for more details.

We need to make one further refinement before resolution can be performed; a CNF clause set needs to be converted into set CNF. A clause set is in set CNF if: (1) none of its clauses are repeated, and (2) none of its clauses contain repeated literals. By throwing out any repeated clauses or literals we can convert a CNF clause set into set CNF. The formulae remain logically equivalent; we are simply removing redundant disjuncts and/or conjuncts.

The resolution proof method is based upon the repeated use of what is called the binary resolution rule.

$$\frac{[p_1,\ldots,p_n,r,p_{n+1},\ldots,p_m] \qquad [q_1,\ldots,q_j,\neg r,q_{j+1},\ldots,q_k]}{[p_1,\ldots,p_n,p_{n+1},\ldots,p_m,q_1,\ldots,q_j,q_{j+1},\ldots,q_k]}$$

Here, given two clauses without repeated literals, $C' \equiv [p_1,\ldots,p_n,r,p_{n+1},\ldots p_m]$ and $C' \equiv [q_1,\ldots,q_j,\neg r,q_{j+1},\ldots q_k]$ say, if C contains a positive literal and C' contains its negation, then we can apply the resolution rule by discarding the pair of literals and merging the remainders to the clause $[p_1,\ldots,p_n,r,p_{n+1},\ldots p_m, q_1,\ldots,q_j,\neg r,q_{j+1},\ldots q_k]$. Note that the merged clause may contain repeated literals; these need to be discarded before this new clause can be resolved against another by further application of the resolution rule. The positive literal and its negation are called a complementary pair, whereas C and C' are called complementary clauses.

We can see that the method is satisfaction preserving; if both C and C' are satisfied in some model M, then at least one literal in each clause must be satisfied in M. Since only one of the complementary pair r and $\neg r$ can be satisfied in M, at least one other literal from either C or C' must be satisfied in M. This literal will feature in the merged clause, hence the merged clause – being a disjunction of literals – will also be satisfied in M.

The general idea behind the resolution proof method is as follows. If we want to show that a formula φ of propositional logic is valid then we use $\neg\varphi$ as input and try to generate an empty clause. If a clause set contains an empty clause – which is always false, regardless of the assignment – then the formula represented by the clause set cannot be satisfied in any model. Therefore if we generate the empty clause from $\neg\varphi$ *via* the satisfaction-preserving resolution method, then φ must be satisfied in all models, hence φ must be valid.

To carry out the method, we first convert $\neg\varphi$ to set CNF. If we can find an empty clause within the clause set, then we are done. If not, we perform the following steps

1. Look for complementary clauses within the clause set. If there are none, we halt and declare that we cannot prove φ valid. For any existing complementary clauses, we apply the resolution rule.
2. If the resultant clause C is the empty clause, we halt and declare φ valid.
3. If C is not the empty clause, but it is a repetition of a clause in the clause set, then we throw C out. If C itself is not repeated, but instead contains repeated literals, then we throw the repeated literals out. We then add C to the clause set and repeat step 1-3 until we halt.

As an example, consider the formula $(\neg p \Rightarrow \neg q) \Rightarrow (q \Rightarrow p)$. We convert its negation to set CNF $[[p,\neg q],[q],[\neg p]]$ using rewrites 3, 4 and 3 and then 5. There are no empty clauses, so we start by resolving $[p,\neg q]$ against $[\neg p]$. This yields $[\neg q]$, which we add to our original clause set to obtain $[[p,\neg q],[q],[\neg p],[\neg q]]$. From here, we resolve $[q]$ against $[\neg q]$, yielding the empty set $[\,]$. We halt and declare $(\neg p \Rightarrow \neg q) \Rightarrow (q \Rightarrow p)$ valid.

When it comes to performing the resolution method on a first-order formula φ involving quantifiers, we need to adjust the conversion to set CNF. The following additional two rewrites allow us to convert φ to NNF.

(Rewrite 10) *as* $\exists x.\neg\varphi$.
(Rewrite 11) $\neg\exists x.\varphi$ as $\forall x.\neg\varphi$.

To get to CNF, we skolemise any existential quantifiers and discard any universal quantifiers. The resultant formula is converted to set CNF as usual. As an example, we convert $\neg(\forall x\exists y.R(x,y) \wedge \forall z\forall w.(R(z,w) \Rightarrow P(z)) \Rightarrow \forall u.P(u))$ to set CNF. We first convert the formula to the NNF $\forall x\exists y.R(x,y) \wedge \forall z\forall w.\ (\neg R(z,w) \vee P(z)) \wedge \exists u.\ \neg P(u)$ using the rewrites 3, 4 and 10. From here, we skolemise $\exists$ and discard $\forall$to obtain $R(x,s(x)) \wedge (\neg R(z,w) \vee P(z)) \wedge \neg P(c)$, where s is a new function symbol and c is a constant. We can write it as the set CNF $\big[\big[R(x,s(x))\big],\big[\neg R(z,w),P(z)\big],\big[\neg P(c)\big]\big]$.

We perform resolution on first-order formulae in set CNF following the same steps outlined above, with the addition that variables can be unified. Since all variables can be universally quantified, we can use unification to create complementary pairs which can then be resolved. Note that variables may need to be relabelled before unification. For example, suppose we have the two clauses $\big[P(x),Q(x)\big]$ and $\big[\neg P(a),R(x)\big]$ where a is a constant and x is a variable. We can unify a with x, however this would affect the x in $R(a)$ which is independent of variable $x$in the first clause. Hence we relabel the variable x in the second clause as y, unify a with x and obtain the clauses $\big[P(a),Q(a)\big]$ and $\big[\neg P(a),R(y)\big]$. We then perform resolution, yielding $\big[Q(a),R(y)\big]$.

Theorem provers that implement resolution-based proof methods for FOL formulae include Prover9 (McCune 2007) – the successor of Otter (McCune 2003) – and Vampire (Riazanov and Voronkov 2002).

4.1.3 Model Building *vs.* Theorem Proving

Again, this section follows from (Blackburn and Bos 2005).

Theorem provers – either tableau or resolution-based – can be used to demonstrate validity, however because of the undecidability of FOL, they are unable to show non-validity. This has severe implications for both the consistency and informativity checking inference tasks.

For example suppose we are performing consistency checking. Consider the discourse *No old lady likes a mystery. Miss M is an old lady. Miss M likes a mystery*[4]. It is obvious that the last sentence is inconsistent with the preceding two sentences. To show this formally, we build a first-order representation for the discourse and check whether the conjunction of the first two sentences implies the negation of the last sentence. (Recall from Section 4.1 that a conclusion ψ is inconsistent with respect to premises $\varphi_1,\ldots,\varphi_n$ if and only if the formula $\varphi_1 \wedge\ldots\wedge \varphi_n \Rightarrow \neg\psi$ is valid.) Hence we need to check whether the following formula is valid.

[4] To write this and the following discourses in PENG, we user-define *mystery* as a singular noun of entity type and atomic structure. We user-define *Miss M* as a singular proper noun with feminine gender and person type, and *irritates* as a finite, singular transitive verb with event structure. Moreover, we define *detective* as a singular noun of person type.

$$\forall x.\ (OlLady(x) \Rightarrow \neg LikesMys(x)) \wedge OlLady(missM) \Rightarrow \neg LikesMys(missM)$$

We can see that the formula is valid; any half-decent theorem prover should be able to prove this. Now suppose our discourse is *No old lady likes a mystery. Miss M is an old lady. Miss M irritates the detective.* Here the last sentence is consistent with the preceding two sentences. To show this formally, again we build a first-order representation for the discourse and check whether the conjunction of the first two sentences implies the negation of the last sentence. Namely, we check the validity of the following formula.

$$\forall x.\ (OlLady(x) \Rightarrow \neg LikesMys(x)) \wedge OlLady(missM) \Rightarrow \neg Irr(missM, detect)$$

We can see that the formula is not valid, however, no theorem prover can show this.

We are faced with a similar problem when it comes to informativity checking. For example, consider the discourse *Every old lady is clever. Miss M is an old lady. Miss M is clever.* The last sentence of this discourse is not informative compared to the two preceding sentences. In order to show this formally, we build a first-order representation and check whether the conjunction of the first two sentences implies the last sentence. (Recall from Section 4.1 that a conclusion ψ is uninformative with respect to premises $\varphi_1, \ldots \varphi_n$ if and only if the formula $\varphi_1 \wedge \ldots \wedge \varphi_n \Rightarrow \psi$ is valid.) Hence we need to check the validity of the following formula.

$$\forall x.\ (OlLady(x) \Rightarrow Clever(x)) \wedge OlLady(missM) \Rightarrow Clever(missM)$$

This is obviously valid and any theorem prover should be able to prove this. Now consider the discourse *Every old lady is clever. Miss M is an old lady. Miss M solves the mystery.* In this case, the last sentence is informative. To show this formally, we need to check the validity of the formula

$$\forall x.\ (OlLady(x) \Rightarrow Clever(x)) \wedge OlLady(missM) \Rightarrow SolveMys(missM)$$

Again we can see that the formula is not valid. Again, no theorem prover can show this.

Full positive checks for consistency and informativity do not exist precisely because of the undecidability of FOL. However, it is possible to conduct partial positive checks for consistency and informativity using a model builder. A model builder takes a formula of FOL and tries to build a finite model that satisfies it. The models that are built are usually small and often a user needs to specify either the size of the domain (*e.g.* 3 elements) or the maximum domain size (*e.g.* a model with at most 20 elements).

For example, suppose we want to run a positive consistency check on the discourse *No old lady likes a mystery. Miss M is an old lady. Miss M irritates the detective.*
We know that if the following formula is not valid

$$\forall x.\ (OlLady(x) \Rightarrow \neg LikesMys(x)) \wedge OlLady(missM) \Rightarrow \neg Irr(missM, detect)$$

then the negation of this formula should be satisfied in at least one model. Hence we know the discourse is consistent if the model builder can find a model for the formula

$$\forall x.\ (OlLady(x) \Rightarrow \neg LikesMys(x)) \wedge OlLady(missM) \wedge \neg Irr(missM, detect)$$

If we specify the model builder to build a model of 2 elements, then it can show consistency by building a model whereby one element is named *detect*, the other *missM*, and *missM* is classified to be a little old lady that doesn't like mysteries and who also irritates the detective. In such a model the latter formula is satisfied. The model builder may build another model

whereby *detect* is classified as a little old lady who doesn't like mysteries, but as long as there is at least one model in which the formula is satisfied, the consistency check returns positive.

As a final example, suppose we want to run a positive informativity check on *Every old lady is clever. Miss M is an old lady. Miss M solves the mystery.* We know the discourse is informative if the model builder can find a model for the negation of the following formula

$$\forall x.\ (OlLady(x) \Rightarrow Clever(x)) \wedge OlLady(missM) \Rightarrow SolveMys(missM)$$

Namely,

$$\forall x.\ (OlLady(x) \Rightarrow Clever(x)) \wedge OlLady(missM) \wedge \neg SolveMys(missM)$$

The model builder can show informativity by building a model whereby an element is named *missM*, and *missM* is classified to be a little old lady who is clever but doesn't solve mysteries. In such a model the latter formula is satisfied. The existence of a model for the negated formula shows that the original formula is not valid and hence the discourse is informative.

4.2 Reasoning in PENG

PENG currently uses the theorem prover Otter and the model builders Mace4 and Satchmo as reasoning tools. Either Otter and Mace4 or Satchmo can be selected during a PENG session. The tools conduct consistency and informativity checks, and allow for question/answering over the input FOL formulae. Otter automatically translates the FOL formulae into set CNF such that it is ready for processing. However Mace4 and Satchmo both rely on a small program at the PENG/reasoner interface which massages the FOL formulae into an acceptable format.

4.2.1 Otter

Otter – the name derived from 'Organised Techniques for Theorem providing and Effective Research' – is a resolution-based theorem prover for FOL with equality developed at the Argonne National Laboratory (McCune 2003). As a resolution-based theorem prover it proves the validity of a FOL formula φ by generating the empty clause from $\neg\varphi$ using a number of resolution rules. It not only applies the binary resolution rule seen in Section 4.1.2, it also applies the following inference rules: hyper-resolution, UR-resolution and binary paramodulation. According to (Wos 2007) a variety of inference rules – such as those featured in Otter – are needed for attacking difficult and disparate problems. We describe the inference rules below. Note that all rules resolve a particular clause of interest – termed a nucleus – with one or more other clauses, termed satellites. The composition of a nucleus and its satellites differs with each rule.

The hyper-resolution inference rule merges two or more clauses. The rule requires that one of the clauses (the nucleus) contains at least one negated literal and the remaining clauses (satellites) contain no negated literals. The rule operates as follows: delete the negated literals of the nucleus along with the matching positive literals from the satellites and merge the remainder. For example, applying hyper-resolution to the set of clauses [[p,q],[$\neg p$,r],[r,s],p] with nucleus [$\neg p$,r] yields [q,r,[r,s]].

The UR-resolution inference rule merges two or more clauses. The 'Unit Resulting' resolution rule requires that the nucleus clause contains at least two literals and the remaining satellite clauses contain exactly one literal each. The rule operates as follows: delete any pairs of literals which simultaneously occur in the nucleus and in any satellite, and merge the remainder. For example, applying UR-resolution to the set of clauses [[p,q,r],q,s,p] with nucleus [p,q,r] yields [r,s].

The binary paramodulation inference rule merges two clauses. The rule requires: (1) that one clause contains at least one literal asserting equality; and (2) that the other clause contains a (possibly negated) literal which features the term on the left-had side of the equality as a sub-expression. The rule operates as follows: delete the equality and merge the remainder along with the literal whose sub-expression has been replaced by the term on the right-hand side of the equality. For example, applying binary paramodulation to the set of clauses [[p,$t = u$],[q,$r(t)$],s] yields [p,q,$r(u)$,s]. (Recall that we are dealing with FOL *with* equality here.)

In contrast to the PENG application, most theorem proving conducted by Otter involves interaction by the user. The user chooses initial conditions, which inference rules to apply and also sets options to control the processing of inferred clauses. PENG however relies on Otter's autonomous mode. In this mode the user inputs a set of formulae and the prover does a simple analysis and decides inference rules and strategies.

Otter has been very stable for a number of years and has been succeeded by Prover9. Prover9 has a number of advantages over Otter: memory consumption is lower, deduction speed is faster, and more inference rules are available. Furthermore, Prover9's autonomous mode is more effective. We refer the reader to (McCune 2007) for details.

4.2.2 Mace4

Mace4 – the name derived from 'Models And Counter Examples' – is a model builder which comes bundled with Prover9 (McCune 2007). The model builder accepts as input a set of FOL formulae which have been restricted and/or formatted in a specific way. We will not list all the (relatively minor) restrictions here, we instead refer the reader to (McCune 2007). Some restrictions worth mentioning are: the builder does not accept function symbols with arity greater than three, nor relation symbols with arity greater than four; neither does the builder accept the natural numbers as constants, instead these are interpreted as elements of the domain. Having accepted the set of formatted FOL formulae, Mace4 then transforms it into a set of propositional formula in CNF. (We do not lose anything during this transformation, since if we have a domain of size n, say, then a universal formula such as $\forall x.P(x)$ is simply the propositional formula $P(a_1) \wedge \ldots \wedge P(a_n)$ where $a_1,\ldots,a_n$ name the elements of the domain.) Once this set has been built, a SAT solver is used to search for a model. (A SAT solver – the name derived from 'SATisfiability' – is designed specifically to solve the propositional satisfiability problem, namely to find a model in which a given propositional formula is satisfiable.) The SAT solver implements a version of the Davis, Putnam, Logemann and Loveland (DPLL) algorithm called ANLDP, derived from 'Argonne National Laboratory – Davis-Putnam' (McCune 1994). ANLDP features some optimisations and efficiency enhancing techniques not present in DPLL, but otherwise the procedure is the same.

Essentially, the DPLL algorithm chooses a literal r from some propositional formula in CNF, assigns its value to be true and then – if possible – propagates this assumption throughout the remaining formulae *via* unit resolution. Unit resolution involves the application of an inference rule which merges two or more clauses. To be applied, the rule requires that the nucleus clause – as discussed in the previous section – consists of a single literal r, say, and that each satellite clause consists of one or more literals. The rule operates as follows: (1) the nucleus is copied into the merged clause; (2) satellite clauses which contain neither r nor $\neg r$ are copied into the merged clause; (3) if one of the satellite clauses contains r, then the entire clause does not appear in the merged clause; and (4) if one of the satellite clauses contains $\neg r$, then this (negative) literal is deleted, and the remainder is copied into the merged clause. For example, suppose we apply unit resolution to the set of clauses $[[p,q],[\neg p,r],[\neg r,s],p]$ with nucleus p. The rule stipulates that $[p,q]$ does not feature in the merged clause. Moreover, we should delete $\neg p$ from clause $[\neg p,r]$ before merging the remainder. Our resultant clause is $[r,[\neg r,s],p]$. After unit resolution has been applied, the DPLL algorithm selects a new literal, assigns it as true and applies unit resolution again. (Following our example, we could perform unit resolution on $[r,[\neg r,s],p]$ with unit clause r, resulting in the clause $[r,s,p]$.) Either the algorithm terminates when it finds an assignment to the literals which satisfies all formulae or it will exhaust all possible decisions for that formula and find it unsatisfiable. Any found models are translated back to first-order models.

Both Mace4 and Prover9 can be found online at (McCune 2005).

4.2.3 Satchmo

Satchmo – the name derived from 'SATisfiability CHecking by MOdel generation' is a model builder developed at the Ludwig Maximilian University of Munich (Abdennadher, Bry et al. 1995; Brüggemann, Bry et al. 1996). It only accepts a set of first-order formulae of the form $antecedent \Rightarrow consequent$. Here *antecedent* is either empty and is thus interpreted as $\top$, or *antecedent* is of the form $p_1 \wedge \ldots \wedge p_n$, where each p_i is a literal for $1 \leq i \leq n$. Furthermore, *consequent* is either empty and is thus interpreted as $\bot$, or *consequent* is of the form $q_1 \vee \ldots \vee q_m$ where each q_j is a literal for $1 \leq j \leq m$. A set of formulae of this form is termed a specification.

Satchmo implements a tableau proof method called Positive Unit Hyper-Resolution (PUHR). (Recall from Section 4.1.1 that a tableau is constructed by breaking complex formulae into atomic formulae – or their negation – through an application of tableau rules. The tableau is said to be saturated when no further rules can be applied.) Essentially, Satchmo constructs a particular tableau – called a PUHR tableau – for a given specification. The saturated tableau is such that the literals of its branches represent the models of the specification. If all branches of the saturated tableau are closed, then the specification is unsatisfiable.

A PUHR tableau is constructed from an initial tableau consisting of a single empty branch using repeated applications of the conjunctive rule – described in Section 4.1.1 – and the PUHR rule outlined below.

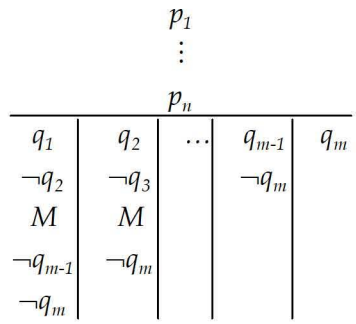
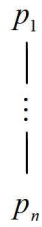
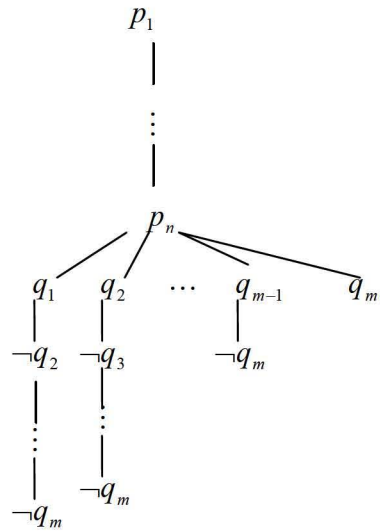


Figure 10: PUHR rule

Recall that all specifications contain formulae of the form $p_1 \wedge \ldots \wedge p_n \Rightarrow q_1 \vee \ldots \vee q_m$. As described in (Brüggemann, Bry et al. 1996) the PUHR rule says that if a branch contains all the literals of a formula's antecedent $p_i, \ldots, p_n$ add m sibling children to the leaf of the branch containing the literals $q_i, \ldots, q_m$ respectively. Then add to each leaf containing q_i the chain of $m - i$ nodes containing the complements of the literal q_j where $j > i$. Namely if we have a tree with the following branch



then add chains of nodes to the branch as follows



The 'complement splitting' process prunes the tableau and guarantees that the generated models are minimal. (Note that a model M of a specification S is said to be minimal if no proper subset of M is a model for S.) As an example, consider the PUHR tableau for the specification $\{p \wedge q \Rightarrow r \vee s \vee t, r \Rightarrow u, s \Rightarrow t, t \Rightarrow u\}$.

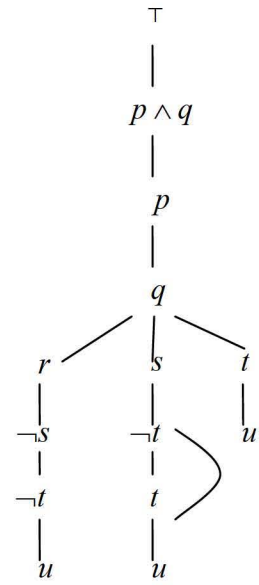


Figure 11: PUHR tableau

We start with an empty branch and attach the antecedent of the first formula as a leaf. We then apply the conjunctive rule which extends the branch with two nodes containing p and q. Following this, we apply the PUHR rule which adds three sibling children to the branch containing the literals r, s and t. We then add to each leaf the chain of nodes containing the relevant complements of the consequent. We next apply the PUHR rule to the second formula of the specification. Since the branch on the left contains the antecedent of the second formula – namely r – we can extend the branch with the (single) consequent u. We don't need to add complements, since there are none. We now move on to apply the rule to the third formula. The middle branch contains the antecedent s, hence we can extend the branch with the consequent t. Again, there are no complements so we add no further branches. We twice apply the PUHR rule to the fourth rule, extending both the middle branch and the branch on the right with the consequent u. Since the middle branch contains a complementary pair, namely t and $\neg t$, we can close the branch. We finally end up with the saturated tableau seen in Figure 10.

The two open branches represent the two possible minimal models for the specification. We can see that the specification is true in a model where p, q, r and u are true. Moreover, we can see that the specification is true in a model where p, q, t and u are true. The literals of the middle branch (p, q, s, t and u) form a superset of the literals of the right branch (p, q, t and u). Forming a non-minimal model, the complement splitting introduced by the PUHR rule has closed this middle branch.

There exist a number of optimisations and efficiency enhancing techniques designed for Satchmo. These techniques determine in which order formulae of the specification are decomposed within the tableau. For more information, see (Abdennadher, Bry et al. 1995; Brüggemann, Bry et al. 1996).

5. Conclusion

This report has described the theoretical underpinnings of the PENG system. We have seen how the restrictions of the controlled natural language allow authors to write text which captures the precision of a formal specification language. The writing process is guided by the automatic generation of look-ahead categories which indicate the possible sentence constructs allowable from the current input. The resulting PENG text looks seemingly informal, but has the same formal properties as the underlying FOL representation. Thus PENG can serve as a high-level interface language to standard FOL theorem provers and model builders. Although PENG is still very much a prototype at this stage, and has a number of issues, the system shows potential.

6. References

Abdennadher, S., F. Bry, et al. (1995). The Theorem Prover Satchmo: Strategies, Heuristics, and Applications – System Description, Technical Report PMS-FB-1995-3, Institute for Informatics, Ludwig Maximilian University of Munich.

Autebert, J.-M., J. Berstel, et al. (1997). Context-Free Languages and Pushdown Automata. Handbook of Formal Languages. G. Rozenberg and A. Salomaa, Springer. **1:** 111-172.

Beckert, B., R. Hähnle, et al. (1996). The Tableau-Based Theorem Prover 3TAP, Version 4.0. In proceedings of the 13th International Conference on Automated Deduction, New Jersey, USA, Springer.

Beckert, B. and J. Posegga. (1997). "LeanTAP Home Page." Retrieved February 2009 from http://www.uni-koblenz.de/~beckert/leantap/.

Blackburn, P. and J. Bos. (1999). "Working with Discourse Representation Theory: An Advanced Course in Computational Semantics." Retrieved February 2009 from http:// www.iccs.inf.ed.ac.uk/~jbos/comsem/book2.html.

Blackburn, P. and J. Bos (2005). Representation and Inference for Natural Language: A First Course in Computational Semantics, CSLI Publications.

Blackburn, P., J. Bos, et al. (2003). "Learn Prolog Now!" Retrieved February 2009 from http://www.learnprolognow.org/.

Blackburn, P. and K. Striegnitz. (2002). "Natural Language Processing Techniques in Prolog." Retrieved February 2009 from http://cs.union.edu/~striegk/courses/nlp-with-prolog/html/.

Boolos, G. (1984). "To be is to be the Value of a Variable (or to be Some Values of Some Variables)." Journal of Philosophy **81**: 430-449.

Brett, A. C. (2000). "An Introduction to Prolog." Retrieved February 2009 from http://web.uvic.ca/~ling48x/ling482/prolog/.

Brüggemann, T., F. Bry, et al. (1996). Satchmo: Minimal Model Generation and Compilation, Technical Report PMS-FB-1996-5, Institute for Informatics, Ludwig Maximilian University of Munich.

Burchardt, A., S. Walter, et al. (2005). "Computational Semantics." Retrieved February 2009 from http://www.coli.uni-saarland.de/projects/milca/courses/comsem/html/.

Chomsky, N. (1956). "Three Models for the Description of Language." IEEE Transactions on Information Theory **2**(3): 113-124.

D'Agostino, M., D. M. Gabbay, et al., Eds. (1999). Handbook of Tableau Methods, Springer.

Deransart, P., A. Ed-Dbali, et al. (1996). Prolog: The Standard: Reference Manual, Springer.

Earley, J. (1970). "An Efficient Context-Free Parsing Algorithm." Communications of the ACM **13**(2): 94-102.

Fitting, M. and R. L. Mendelsohn (1998). First-Order Modal Logic, Kluwer Academic Publishers.

Fuchs, N. E., H. F. Hofmann, et al. (1994). Specifying Logic Programs in Controlled Natural Language, Technical Report 94.17, Department of Information Technology, University of Zürich.

Gal, A., G. Lapalme, et al. (1991). Prolog for Natural Language Processing, Wiley.

Gazdar, G. and C. Mellish (1990). Natural Language Processing in PROLOG: An Introduction to Computational Linguistics, Addison-Wesley.

Gilbert, P. (1966). "On the Syntax of Algorithmic Languages." Journal of the Association for Computing Machinery **13**(1): 90-107.

Jurafsky, D. and J. H. Martin (2000). Speech and Language Processing, Prentice Hall.

Kamp, H. and U. Reyle (1993). From Discourse to Logic, Kluwer.

Kasami, T. (1965). An Efficient Recognition and Syntax Analysis Algorithm for Context-Free Languages, Technical Report AF CRL-65-758, Air Force Cambridge Research Laboratory.

Leitch, A. (1997). The Resolution Calculus, Springer.

Linnebo, Ø. (2005). Plural Quantification. The Stanford Encyclopedia of Philosophy. E. N. Zalta, Stanford University.

Longley, J. and I. Stark. (2002). "Top-Down and Bottom-Up Parsing." Retrieved February 2009 from http://www.inf.ed.ac.uk/teaching/courses/cs2/LectureNotes/CS2Ah/LangProc/lp9.pdf.

McCune, W. (1994). A Davis-Putnam Program and its Application to Finite First-Order Model Search: Quasi-Group Existence Problems, Technical Report ANL/MCS-TM-194, Argonne National Laboratory.

McCune, W. (2003). "Otter and Mace2." Retrieved February 2009 from http://www.cs.unm.edu/~mccune/otter/.

McCune, W. (2005). "Son of BirdBrain II." Retrieved February 2009 from http://www.cs.unm.edu/~mccune/sobb2/.

McCune, W. (2007). "Prover9 and Mace4." Retrieved February 2009 from http://www.cs.unm.edu/~mccune/mace4/.

McKay, T. (2006). A Formal Language with Non-Distributive Plurals: Preliminary Considerations. Plural Predication, Oxford University Press: 5-17.

Meurers, D. (2003). Introduction to Computational Linguistics I, Lecture Notes 684.01, Department of Linguistics, Ohio State University.

Mitkov, R. (2003). Anaphora Resolution. The Oxford Handbook of Computational Linguistics. R. Mitkov, Oxford University Press: 266-284.

Nugues, P. (2006). An Introduction to Language Processing with Perl and Prolog, Springer.

Pelletier, F. J. (1986). "Seventy-Five Problems for Testing Automatic Theorem Provers." Journal of Automated Reasoning **2**(2): 191-216.

Rayo, A. (2002). "Word and Objects." Noûs **36**(3): 436-464.

Riazanov, A. and A. Voronkov (2002). "The Design and Implementation of VAMPIRE." AI Communications **15**(2): 91-110.

Schwertel, U. (2005). Plural Semantics for Natural Language Understanding: A Computational Proof-Theoretic Approach. Faculty of Arts, University of Zurich.

Schwitter, R. (2003). Incremental Chart Parsing with Predictive Hints. In proceedings of the Australasian Language Technology Workshop, University of Melbourne, Australia.

Schwitter, R. (2004a). Dynamic Semantics for a Controlled Natural Language. In proceedings of the 15th International Workshop on Database and Expert Systems Applications, Zaragoza, Spain.

Schwitter, R. (2004b). Representing Knowledge in Controlled Natural Language: A Case Study. In proceedings of the 8th International Conference on Knowledge-Based Intelligent Information and Engineering Systems, Wellington, New Zealand.

Schwitter, R. (2007a). Grammar Rules in PENG: Email.

Schwitter, R. (2007b). Illegal Words in PENG: Email.

Schwitter, R. (2007c). "PENG Online." Retrieved February 2009 from http://www.ics.mq.edu.au/~peng/PengEditor.html.

Schwitter, R. and A. Ljungberg (2002). How to Write a Document in Controlled Natural Language. In proceedings of the 7th Australasian Document Computing Symposium, Sydney, Australia.

Schwitter, R., A. Ljungberg, et al. (2003). ECOLE: A Look-Ahead Editor for a Controlled Language. In proceedings of the joint conference combining the 8th International Workshop of the European Association for Machine Translation and the 4th Controlled Language Application Workshop, Dublin City University, Ireland.

Schwitter, R. and M. Tilbrook (2004a). Controlled Natural Language meets the Semantic Web. In proceedings of the Australasian Language Technology Workshop, Macquarie University, Australia.

Schwitter, R. and M. Tilbrook (2004b). Dynamic Semantics at Work. In proceedings of the International Workshop on Logic and Engineering of Natural Language Semantics, Kanazawa, Japan.

Schwitter, R. and M. Tilbrook (2006). Controlled Language Processing, Deliverable 2006-CLP-1, Centre for Language Technology.

Sterling, L. and E. Y. Shapiro (1994). The Art of Prolog: Advanced Programming Techniques, MIT Press.

Stone, M. (2002). Knowledge Representation. A Handbook for Language Engineers. A. Farghaly, CSLI.

Wirén, M. (1989). Interactive Incremental Chart Parsing. In proceedings of the Fourth Conference of the European Chapter of the Association for Computational Linguistics Manchester, England.

Wirén, M. (1994). Minimal Change and Bounded Incremental Parsing. In proceedings of the 15th International Conference on Computational Linguistics, Kyoto, Japan.

Wos, L. (2007). "A Summary of Inference Rules Used by Argonne's Automated Deduction Software." Retrieved February 2009 from http://www-unix.mcs.anl.gov/AR/inf_rules.html.

Younger, D. (1967). "Recognition of Context-Free Languages in Time n^3." Information and Control **10**(2): 189-208.

THIS PAGE HAS BEEN
INTENTIONALLY
LEFT BLANK

Appendix A: Sentence Structure

Sentences in PENG are of the following form. Note that this is in no way an exhaustive list; we simply want to give the reader a 'feel' for the language. We also show some of the restrictions placed on the grammar, intentional or otherwise. Words in italics represent content words, whereas words in normal font represent function words.

[pn,det,card,conn:[If]]
Agatha [aux:[does],cop:[is],relpro:[that,who],v]
Agatha dances [adv,conn:[and,or],prep:[around,at,by,in,on,with],fs:[.]]

Here the word *dances* is a user-defined intransitive verb.

Agatha dances enthusiastically [conn:[and,or],fs:[.]]
Agatha dances and [aux:[does],cop:[is],v]
Agatha dances or [aux:[does],cop:[is],v]

The grammar does not allow us to write *Agatha dances then*, however we can write *If Agatha dances then*.

Agatha dances around [conn:[and,or],fs:[.]]
Agatha dances at [det,pn,var]
Agatha dances at a [adj,n]
Agatha dances at a *famous nightclub* [conn:[and,or],prep:[of],pn,relpro:[that,which],var,fs:[.]]
Agatha dances at a *famous nightclub* and [aux:[does],cop:[is],v]
Agatha dances at a *famous nightclub* or [aux:[does],cop:[is],v]
Agatha dances at a *famous nightclub* of [det,pn,var]
Agatha dances at a *famous nightclub* of a *corrupt businessman* [conn:[and,or],prep:[of],pn,relpro:[that,who],var,fs:[.]]
Agatha dances at a *famous nightclub* of a *dangerous city* [conn:[and,or],prep:[of],pn,relpro:[that,which],var,fs:[.]]
Agatha dances at a *famous nightclub* of a *tumultuous time* [conn:[and,or],prep:[of],pn,relpro:[that,which],var,fs:[.]]
Agatha dances at a *famous nightclub Fifty-Four* [conn:[and,or],prep:[of],relpro:[that,which],fs:[.]]
Agatha dances at a *famous nightclub* that [aux:[does],cop:[is],det,pn,var,v]
Agatha dances at a *famous nightclub* which [aux:[does],cop:[is],det,pn,var,v]
Agatha dances at a *famous nightclub X1* [conn:[and,or],prep:[of],relpro:[that,which],fs:[.]]
Agatha dances at a *famous nightclub.*
Agatha dances at a *tumultuous time* [conn:[and,or],prep:[of],pn,relpro:[that,which],var,fs:[.]]
Agatha dances at all *famous nightclubs* [conn:[and,or],relpro:[that,which],fs:[.]]
Agatha dances at all *famous nightclubs* that [aux:[do],cop:[are],det,pn,var,v]
Agatha dances at all *famous nightclubs* which [aux:[do],cop:[are],det,pn,var,v]
Agatha dances at an *famous nightclub* [conn:[and,or],prep:[of],pn,relpro:[that,which],var,fs:[.]]

It is not required that adjectives or nouns following the determiner *an* start with a vowel.

Agatha dances at every *famous nightclub* [conn:[and,or],prep:[of],pn,relpro:[that,which],var,fs:[.]]
Agatha dances at no *famous nightclub* [conn:[and,or],prep:[of],pn,relpro:[that,which],var,fs:[.]]
Agatha dances at no *famous nightclubs* [conn:[and,or],relpro:[that,which],fs:[.]]
Agatha dances at the *famous nightclub* [conn:[and,or],prep:[of],pn,relpro:[that,which],var,fs:[.]]
Agatha dances at the *famous nightclubs* [conn:[and,or],relpro:[that,which],fs:[.]]
Agatha dances at *Fifty-Four* [conn:[and,or],relpro:[that,which],fs:[.]]
Agatha dances at *Friday* [conn:[and,or],relpro:[that,which],fs:[.]]
Agatha dances at *X1* [conn:[and,or],relpro:[that,which],fs:[.]]

The nouns following the function word *at* – in our examples these are *nightclub*, *businessman*, *city* and *time* – must be of type entity or time. For example, since *smiles* is base-defined as an intransitive verb, we cannot write *smiles at Charles, smiles at all bartenders*, nor *smiles at a B–list celebrity,* where *Charles*, *bartenders* and *celebrity* are of type person. Note that since the word *shouts* has not been base-defined, we can write *shouts at Charles*, *shouts at bartenders, etc.* but here *shouts at* must be user-defined as a prepositional transitive verb and we lose the ability to user-define *shouts* as an intransitive verb.

Agatha dances by [det,pn,var]
Agatha dances by a *famous nightclub* [conn:[and,or],prep:[of],pn,relpro:[that,which],var,fs:[.]]

The nouns following the function word *by* must be of type entity. We cannot write *dances by Charles* nor *dances by a B–list celebrity.* Furthermore we cannot write the ungrammatical strings *dances by Friday* or *dances by a tumultuous time.*

Agatha dances in a *famous nightclub* [conn:[and,or],prep:[of],pn,relpro:[that,which],var,fs:[.]]
Agatha dances in a *tumultuous time* [conn:[and,or],prep:[of],pn,relpro:[that,which],var,fs:[.]]
Agatha dances on a *famous nightclub* [conn:[and,or],prep:[of],pn,relpro:[that,which],var,fs:[.]]
Agatha dances on a *tumultuous time* [conn:[and,or],prep:[of],pn,relpro:[that,which],var,fs:[.]]

The nouns following the function words *in* or *on* must be of type entity or time. Note we cannot write *falls on a B–list celebrity* or *votes in Charles*, where *falls* and *votes* are both base-defined intransitive verbs.

Agatha dances with [det,pn,var]
Agatha dances with a *tattered feather boa* [conn:[and,or],prep:[of],pn,relpro:[that,which],var,fs:[.]]
Agatha dances with a *B-list celebrity* [conn:[and,or],prep:[of],pn,relpro:[that,who],var,fs:[.]]

The nouns following the function word *with* must be of type entity or person. We cannot write the ungrammatical string *dances with a tumultuous time*. Moreover, we cannot combine any intransitive verb (either base or user-defined) with the prepositions *about*, *for*, *like*, *of*, *over*, *than* and *to*. Thus no *votes for, dreams about, thinks about, thinks of, stands over, sleeps like, approves of, runs to, falls over, returns for, appears to, works for, runs like, exists to, acts like, waits for, lives like, resigns over, sings like, sleeps over, stands about, differs over, returns to, continues to, cries like, etc.* We cannot turn any of these into prepositional transitives verbs since the intransitive verbs are already defined. We can user-define prepositional transitive verbs such as *talks for*, *walks about* and *looks like*, but we will then only be able to define *talks*, *walks* and *looks* as transitive, not intransitive verbs.

Agatha dances.
Agatha drives [det,pn,var]

Here *drives* is a transitive verb, as are the words *has*, *likes* and *hates* (which are used later). Note that we cannot write a transitive verb followed by the function word *that*, hence no *admits that, announces that, argues that, concludes that, confirms that, decides that, demands that, etc.*

Agatha drives a [adj,n]
Agatha drives a *yellow sportscar* [adv,conn:[and,or],prep:[around,at,by,in,on,of,with],pn,relpro:[that,which],var,fs:[.]]

Agatha drives a *yellow sportscar fast* [conn:[and,or],fs:[.]]
Agatha drives a *yellow sportscar Porsche* [adv,conn:[and,or],prep:[around,at,by,in,on,of,with],relpro:[that,which],fs:[.]]

We cannot insert a proper noun between an adjective and a noun, hence we cannot write *drives a yellow Porsche sportscar* nor *has a brown Gucci handbag*.

Agatha drives a *B-list celebrity* [adv,conn:[and,or],prep:[around,at,by,in,on,of,with],pn,relpro:[that,who],var,fs:[.]]
Agatha has a *pleasant day* [adv,conn:[and,or],prep:[around,at,by,in,on,of,with],pn,relpro:[that,which],var,fs:[.]]
Agatha drives all *B-list celebrities* [adv,conn:[and,or],prep:[around,at,by,in,on,with],relpro:[that,who],fs:[.]]
Agatha drives all *afternoon* [adv,conn:[and,or],prep:[around,at,by,in,on,with],relpro:[that,which],fs:[.]]
Agatha likes New York [adv,conn:[and,or],prep:[around,at,by,in,on,with],relpro:[that,which],fs:[.]]
Agatha hates Charles [adv,conn:[and,or],prep:[around,at,by,in,on,with],relpro:[that,who],fs:[.]]
Agatha drives X1 [adv,conn:[and,or],prep:[around,at,by,in,on,with],relpro:[that,which,who],fs:[.]]

Note we cannot combine any transitive verb determiner noun sequence nor transitive verb proper noun sequence (either base or user-defined) with the prepositions *about*, *for*, *like*, *over*, *than* and *to*. Thus no *approached X about*, *adds X to*, *allows X to*, *arranges X to*, *brings X to*, *blames X for*, *buys X for*, *creates X to*, *converts X to*, *compensates X for*, *describes X to*, *drives X to*, *enables X to*, *explains X to*, *finds X for*, *fixes X for*, *gets X for*, *has X for*, *makes X for*, *meets X to*, *needs X for*, *orders X to*, *prefers X over*, *reads X to*, *receives X for*, *reports X to*, *searches X for*, *sells X to*, *sends X to*, *takes X to* and *wants X to*, *etc.* Here *X* represents either a noun phrase (*i.e.* a determiner adjective noun sequence) or a proper noun. We cannot turn any of these into prepositional ditransitive verbs since the transitive verbs are already defined.

We can convert any transitive verb into a prepositional transitive verb as long as we only use the prepositions *around*, *about*, *at*, *for*, *like*, *of*, *over*, *than*, and *to*. For some unknown reason the lexical editor will accept prepositional transitive verbs using the prepositions *by*, *in*, *on* and *with*, but the PENG editor will not let us enter them.

Agatha adapts [prep:[*to*]]
Agatha adapts to [det,pn,var]

Here *adapts to* is a prepositional transitive verb. For some unknown reason the base-defined prepositional transitive verb *participates in* causes an error.

Agatha adapts to a [adj,n]
Agatha adapts to a *B-list celebrity* [adv,conn:[and,or],prep:[around,at,by,in,on,of,with],pn,relpro:[that,who],var,fs:[.]]
Agatha adapts to a *famous nightclub* [adv,conn:[and,or],prep:[around,at,by,in,on,of,with],pn,relpro:[that,which],var,fs:[.]]
Agatha adapts to a *tumultuous time* [adv,conn:[and,or],prep:[around,at,by,in,on,of,with],pn,relpro:[that,which],var,fs:[.]]
Agatha adapts to Charles [adv,conn:[and,or],prep:[around,at,by,in,on,with],relpro:[that,who],fs:[.]]
Agatha adapts to New York [adv,conn:[and,or],prep:[around,at,by,in,on,with],relpro:[that,which],fs:[.]]
Agatha adapts to Friday [adv,conn:[and,or],prep:[around,at,by,in,on,with],relpro:[that,which],fs:[.]]
Agatha adapts to X1 [adv,conn:[and,or],prep:[around,at,by,in,on,with],relpro:[that,which,who],fs:[.]]

There are a number of words such as *appears*, *goes*, *stands*, *wakes*, *etc.* which are base-defined as intransitive verbs and are also used in base-defined prepositional transitive verbs, *i.e. appears in*, *goes to*, *stands in*, *wakes up*, *etc.* Below lists the look-ahead categories generated by *appears* and *appears in*.

Agatha appears [adv,conn:[and,or],prep:[around,at,by,in,on,with],fs:[.]]
Agatha appears in [det,pn,var]
Agatha appears in a *famous nightclub* [conn:[and,or],prep:[of],pn,relpro:[that,which],var,fs:[.]]
Agatha appears in a *tumultuous time* [conn:[and,or],prep:[of],pn,relpro:[that,which],var,fs:[.]]

The word *appears* is initially treated as an intransitive verb. Note however, that if the preposition *in* follows, then the look-ahead categories for *appears in* do not unfold as for a prepositional transitive verb; they unfold as for an intransitive verb. Note also that the preposition determines the noun type. We can have *adapts to* followed by a noun phrase or proper noun of type person, entity or time; but we cannot have *appears in* followed by a noun phrase or proper noun of type person. Nor, for example, could we have *agrees with* followed by a noun phrase or proper noun of type time. A different effect is produced by the word *turns* which is base-defined as a transitive verb and also used in the base-defined prepositional transitive verb *turns around*. Below lists the look-ahead categories generated by *turns* and *turns around*.

Agatha turns [det,prep:[around],pn,var]
Agatha turns around [det,pn,var]
Agatha turns around a *famous nightclub* [adv,conn:[and,or],prep:[around,at,by,in,on,of,with],pn,relpro:[that,which],var,fs:[.]]
Agatha turns around a *tumultuous time* [adv,conn:[and,or],prep:[around,at,by,in,on,of,with],pn,relpro:[that,which],var,fs:[.]]
Agatha turns around a *B-list celebrity* [adv,conn:[and,or],prep:[around,at,by,in,on,of,with],pn,relpro:[that,who],var,fs:[.]]

Note that *turns* is initially treated as a transitive verb with the preposition *around* added to the look-ahead category. As before, the look-ahead categories for *turns around* do not unfold as for a prepositional transitive verb; they unfold as for a transitive verb.

Agatha tells [det,pn,var]

Here *tells* is a ditransitive verb, as is the word *regards* (which is used later). Note we can convert a ditransitive verb to a prepositional transitive verb, hence we can user-define *tells to*, *offers to*, *sends for*, and *sticks to*.

Agatha tells a [adj,n]
Agatha tells a *B-list celebrity* [det,prep:[of],pn,relpro:[that,who],var]
Agatha tells a *B-list celebrity* of [det,pn,var]
Agatha tells a *B-list celebrity Max* [adv,conn:[and,or],det,prep:[around,at,by,in,on,of,with],pn,relpro:[that,who],var,fs:[.]]
Agatha tells a *B-list celebrity Friday* [adv,conn:[and,or],det,prep:[around,at,by,in,on,of,with],pn,relpro:[that,which,who],var, fs:[.]]
Agatha tells a *scary story* [det,prep:[of],pn,relpro:[that,which],var]
Agatha tells a *scary story The Ghost of Dreadsbury* [adv,conn:[and,or],det,prep:[around,at,by,in,on,of,with],pn, relpro:[that,which],var,fs:[.]]
Agatha tells a *scary story Max* [adv,conn:[and,or],det,prep:[around,at,by,in,on,of,with],pn,relpro:[that,which,who],var,fs:[.]]
Agatha regards a *sunny day* [det,prep:[of],pn,relpro:[that,which],var]
Agatha tells Max [det,pn,relpro:[that,who],var]
Agatha regards Friday [det,pn,relpro:[that,which],var]
Agatha regards X1 [det,pn,relpro:[that,which,who],var]

The base-defined ditransitive verb *sends* is also base-defined as a transitive verb. Below lists the look-ahead categories generated by *sends*.

Agatha sends [det,pn,var]
Agatha sends a [adj,n]
Agatha sends a *B-list celebrity* [adv,conn:[and,or],det,prep:[around,at,by,in,on,of,with],pn,relpro:[that,who],var,fs:[.]]
Agatha sends a *lascivious postcard* [adv,conn:[and,or],det,prep:[around,at,by,in,on,of,with],pn,relpro:[that,which],var,fs:[.]]
Agatha sends Max [adv,conn:[and,or],det,prep:[around,at,by,in,on,with],pn,relpro:[that,who],var,fs:[.]]
Agatha sends New York [adv,conn:[and,or],det,prep:[around,at,by,in,on,with],pn,relpro:[that,which],var,fs:[.]]

It looks like the look-ahead categories generated for ditransitive and transitive verbs have been combined to form the look-ahead categories for *sends*.

Agatha hands [det,pn,var]

Here *hands over* is a prepositional ditransitive verb.

Agatha hands a [adj,n]
Agatha hands a *troublesome child* [prep:[of,*over*],pn,relpro:[that,who],var]
Agatha hands a *troublesome child* of [det,pn,var]
Agatha hands a *troublesome child* of an [adj,n]
Agatha hands a *troublesome child* of an *unemployed actor* [prep:[of,*over*],pn,relpro:[that,who],var]
Agatha hands a *troublesome child* of an *unemployed actor over* [det,pn,var]
Agatha hands a *troublesome child* of an *unemployed actor over* a [adj,n]
Agatha hands a *troublesome child* of an *unemployed actor over* a *fence* [adv,conn:[and,or],
prep:[around,at,by,in,of,on,with],pn,relpro:[that,which],var,fs:[.]]
Agatha hands a *troublesome child* of an *unemployed actor Max* [prep:[of,*over*],relpro:[that,who]]
Agatha hands a *troublesome child* of an *unemployed actor* that [aux:[does],cop:[is],det,pn,var,v]
Agatha hands a *troublesome child* of an *unemployed actor* that does [neg:[not]]
Agatha hands a *troublesome child* of an *unemployed actor* that does not [v]
Agatha hands a *troublesome child* of an *unemployed actor* that does not *dance* [adv,conn:[and,or],
prep:[around,at,by,in,on,*over*,with]]
Agatha hands a *troublesome child over* [det,pn,var]
Agatha hands a *troublesome child Maxine* [prep:[of,*over*],relpro:[that,who]]
Agatha hands a *best-selling novel* [prep:[of,*over*],pn,relpro:[that,which],var]
Agatha hands Maxine [prep:[*over*],relpro:[that,who]]
Agatha hands The Deathly Hallows [prep:[*over*],relpro:[that,which]]
Agatha hands X1 [prep:[*over*],relpro:[that,which,who]]

The base-defined ditransitive verb *gives* is also used in the base-defined prepositional ditransitive verb *gives to*. Below lists the look-ahead categories generated by *gives to*.

Agatha gives [det,pn,var]
Agatha gives a [adj,n]
Agatha gives a *troublesome child* [det,prep:[of,*to*],pn,relpro:[that,who],var]
Agatha gives a *best-selling novel* [det,prep:[of,*to*],pn,relpro:[that,which],var]
Agatha gives Maxine [det,prep:[*to*],pn,relpro:[that,who],var]
Agatha gives X1 [det,prep:[*to*],pn,relpro:[that,which,who],var]

Again, it looks like the look-ahead categories generated for ditransitive and prepositional ditransitive verbs have been combined to form the look-ahead categories for *gives to*.

Agatha does [neg:[not]]
Agatha does not [v]
Agatha is [adj,det:[a,an,the],neg:[not],prep:[at,by,in,on,with],pn,var]
Agatha is *tall* [adv,conn:[and,or],prep:[around,at,by,in,on,with],fs:[.]]
Agatha is a [adj,n]
Agatha is a *tall woman* [adv, conn:[and,or],prep:[around,at,by,in,on,with],pn,relpro:[that,who],var,fs:[.]]
Agatha is a *tall woman* that [aux:[does],det,pn,var,v]
Agatha is a *tall woman* who [aux:[does],det,pn,var,v]
Agatha is not [adj,det:[a,an,the],neg:[not],prep:[at,by,in,on,with],pn,var]
Agatha is at [det,pn,var]
Agatha is by [det,pn,var]
Agatha is in [det,pn,var]
Agatha is on [det,pn,var]
Agatha is with [det,pn,var]
Agatha is *Ms. Dreadsbury* [adv,conn:[and,or],prep:[around,at,by,in,on,with],relpro:[that,who],fs:[.]]
Agatha is *X1* [adv,conn:[and,or],prep:[around,at,by,in,on,with],relpro:[that,which,who],fs:[.]]
Agatha that [aux:[does],cop:[is],det,pn,var,v]
Agatha that is [adj,det:[a,an,the],neg:[not],prep:[at,by,in,on,with],pn,var]
Agatha who [aux:[does],cop:[is],det,pn,var,v]
New York [aux:[does],cop:[is],relpro:[that,which],v]
Friday [aux:[does],cop:[is],relpro:[that,which],v]
A [adj,n]
A *famous nightclub* [aux:[does],cop:[is],prep:[of],pn,relpro:[that,which],var,v]
All [adj,n]
All *famous nightclubs* [aux:[do],cop:[are],relpro:[that,which],v]
Eight [adj,n]
Eight *B-list celebrities* [dis:[each,together],relpro:[that,who]]
Eight *B-list celebrities* each [aux:[do],cop:[are],prep:[of],relpro:[that,who],v]
Eight *B-list celebrities* together [aux:[do],cop:[are],prep:[of],relpro:[that,who],v]
Eight *B-list celebrities* that [aux:[do],cop:[are],det,pn,var,v]
Eight *B-list celebrities* who [aux:[do],cop:[are],det,pn,var,v]
Eight *best-selling novels* [dis:[each,together],relpro:[that,which]]
Eight *sunny days* [dis:[each,together],relpro:[that,which]]
If [det,pn,var]
If a *B-list celebrity* [aux:[does],cop:[is],prep:[of],pn,relpro:[that,who],var,v]
If a *B-list celebrity* is [adj,det:[a,an,the],neg:[not],prep:[at,by,in,on,with],pn,var]
If a *B-list celebrity* is *unemployed* [adv,conn:[and,or,then],prep:[around,at,by,in,on,with]]
If a *B-list celebrity* is *unemployed* then [det,pn,var]
If a *B-list celebrity* of [det,pn,var]
If a *B-list celebrity* of a [adj,n]
If a *B-list celebrity* of a *trashy sit-com* [aux:[does],cop:[is],prep:[of],pn,relpro:[that,which],var,v]
If a *B-list celebrity Max* [aux:[does],cop:[is],prep:[of], relpro:[that,who],v]
If a *B-list celebrity X1* [aux:[does],cop:[is],prep:[of],relpro:[that,who],v]
If a *B-list celebrity dances* [adv,conn:[and,or,then],prep:[around,at,by,in,on,with]]
If a *B-list celebrity drives* [det,pn,var]
If a *best-selling novel* [aux:[does],cop:[is],prep:[of],pn,relpro:[that,which],var,v]
If a *sunny day* [aux:[does],cop:[is],prep:[of],pn,relpro:[that,which],var,v]
If all *sunny days* [aux:[do],cop:[are],relpro:[that,which],v]
If *Agatha* [aux:[does],cop:[is],relpro:[that,who],v]
If *Agatha dances* [adv,conn:[and,or,then],prep:[around,at,by,in,on,with]]
If *X1* [aux:[does],cop:[is],relpro:[that,who,which],v]

Below we list the look-ahead categories for questions posed in PENG.

[wh_quest:[Who,What,Where,When,How],yn_quest:[Does,Is]]
Who [aux:[does],cop:[is],v]
Who does [det:[a,an,every,no,the],neg:[not],pn,var]

Note that we cannot construct the ungrammatical string *Who does all*.

Who does a [adj,n]
Who does a *B-list celebrity* [prep:[of],pn,relpro:[that,who],var,v]
Who does a *famous nightclub* [prep:[of],pn,relpro:[that,which],var,v]
Who does a *sunny day* [prep:[of],pn,relpro:[that,which],var,v]
Who does not [v]
Who does *Agatha* [relpro:[that,who],v]
Who does *Agatha avoid* [adv,conn:[and,or],prep:[around,at,by,in,on,with],qm:[?]]
Who does *X1* [relpro:[that,which,who],v]
Who is [adj,det:[a,an,the],neg:[not],prep:[at,by,in,on,with],pn,var]
Who is *tall* [adv,conn:[and,or],prep:[around,at,by,in,on,with],qm:[?]]
Who is a [adj,n]
Who is a *B-list celebrity* [adv,conn:[and,or],prep:[around,at,by,in,on,with],pn,relpro:[that,who],var,qm:[?]]
Who is a *famous nightclub* [adv,conn:[and,or],prep:[around,at,by,in,on,with],pn,relpro:[that,which],var,qm:[?]]
Who is a *sunny day* [adv,conn:[and,or],prep:[around,at,by,in,on,with],pn,relpro:[that,which],var,qm:[?]]
Who is not [adj,det:[a,an,the],prep:[at,by,in,on,with],pn,var]
Who is not *tall* [adv,conn:[and,or],prep:[around,at,by,in,on,with],qm:[?]]
Who is not a [adj,n]
Who is not a *B-list celebrity* [adv,conn:[and,or],prep:[around,at,by,in,on,with],pn,relpro:[that,who],var,qm:[?]]
Who is *Agatha* [adv,conn:[and,or],prep:[around,at,by,in,on,with],relpro:[that,who],qm:[?]]
Who is *X1* [adv,conn:[and,or],prep:[around,at,by,in,on,with],relpro:[that,which,who],qm:[?]]
Who *dances* [adv,conn:[and,or],prep:[around,at,by,in,on,with],qm:[?]]
What [aux:[does],cop:[is],v]
What does [det:[a,an,every,no,the],neg:[not],pn,var]
What is [adj,det:[a,an,the],neg:[not],prep:[at,by,in,on,with],pn,var]
What *affects* [det,pn,var]
Where [aux:[does],cop:[is]]
Where does [det:[a,an,every,no,the],pn,var]
Where does a [adj,n]
Where does a *B-list celebrity* [neg:[not],prep:[of],pn,relpro:[that,who],var,v]
Where does a *B-list celebrity* not [v]
Where does a *famous nightclub* [neg:[not],prep:[of],pn,relpro:[that,which],var,v]
Where does a *sunny day* [neg:[not],prep:[of],pn,relpro:[that,which],var,v]
Where is [det,pn,var]
Where is a [adj,n]
Where is a *B-list celebrity* [adj,det,neg:[not],prep:[at,by,in,of,on,with],pn,relpro:[that,who],var]
Where is a *famous nightclub* [adj,det,neg:[not],prep:[at,by,in,of,on,with],pn,relpro:[that,which],var]
Where is a *sunny day* [adj,det,neg:[not],prep:[at,by,in,of,on,with],pn,relpro:[that,which],var]
Where is *Agatha* [adj,det,neg:[not],prep:[at,by,in,of,on,with],pn,relpro:[that,who],var]
Where is *New York* [adj,det,neg:[not],prep:[at,by,in,of,on,with],pn,relpro:[that,which],var]
Where is *X1* [adj,det,neg:[not],prep:[at,by,in,of,on,with],pn,relpro:[that,which,who],var]
When [aux:[does],cop:[is]]
When does [det:[a,an,every,no,the],pn,var]
When is [det,pn,var]

When is *Agatha* [adj,det,neg:[not],prep:[at,by,in,of,on,with],pn,relpro:[that,who],var]
When is *Agatha* a [adj,n]
When is *Agatha* a *B-list celebrity* [conn:[and,or],prep:[at,by,in,on,with],pn,relpro:[that,who],var,qm:[?]]
When is *Agatha* not [adj,det,prep:[at,by,in,of,on,with],pn,var]
When is *Agatha* not a [adj,n]
When is *Agatha* not a *B-list celebrity* [prep:[at,by,in,on,with],pn,relpro:[that,who],var]
How [aux:[does],cop:[is]]
How does [det:[a,an,every,no,the],pn,var]
How is [det,pn,var]

Note that the grammar does not allow us to ask questions such as *Where is New York?*, *Where is the famous nightclub?*, *How is Agatha?*, *How is the B–list celebrity?* nor *When is the wedding?* Furthermore we cannot ask questions beginning *Who do all*, *What do all*, *Where do all*, *When do all* and *How do all*, nor *Who are all*, *What are all*, *Where are all*, *When are all* and *How are all*.

Does [det:[a,an,every,no,the],pn,var]
Does a [adj,n]
Does a *B-list celebrity* [neg:[not],prep:[of],pn,relpro:[that,who],var,v]
Does a *famous nightclub* [neg:[not],prep:[of],pn,relpro:[that,which],var,v]
Does a *sunny day* [neg:[not],prep:[of],pn,relpro:[that,which],var,v]
Does *Agatha* [neg:[not],relpro:[that,who],v]
Does *Agatha* not [v]
Does *Agatha dance* [adv,conn:[and,or],prep:[around,at,by,in,on,with],qm:[?]]
Does *X1* [neg:[not],relpro:[that,which,who],v]
Is [det:[a,an,every,no,the],pn,var]
Is a [adj,n]
Is a *B-list celebrity* [adj,det,neg:[not],prep:[at,by,in,of,on,with],pn,relpro:[that,who],var]
Is a *famous nightclub* [adj,det,neg:[not],prep:[at,by,in,of,on,with],pn,relpro:[that,which],var]
Is a *sunny day* [adj,det,neg:[not],prep:[at,by,in,of,on,with],pn,relpro:[that,which],var]
Is *Agatha* [adj,det,neg:[not],prep:[at,by,in,of,on,with],pn,relpro:[that,who],var]
Is *Agatha tall* [adv,conn:[and,or],prep:[around,at,by,in,on,with],qm:[?]]
Is *Agatha* a [adj,n]
Is *Agatha* a *tall person* [adv,conn:[and,or],prep:[around,at,by,in,on,with],pn,relpro:[that,who],var,qm:[?]]
Is *Agatha Max* [adv,conn:[and,or],prep:[around,at,by,in,on,with],relpro:[that,who],qm:[?]]
Is *X1* [adj,det,neg:[not],prep:[at,by,in,of,on,with],pn,relpro:[that,which,who],var]

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION
DOCUMENT CONTROL DATA

1. PRIVACY MARKING/CAVEAT (OF DOCUMENT)

2. TITLE

Processable English:
The Theory Behind the PENG System

3. SECURITY CLASSIFICATION (FOR UNCLASSIFIED REPORTS THAT ARE LIMITED RELEASE USE (L) NEXT TO DOCUMENT CLASSIFICATION)

Document	(U)
Title	(U)
Abstract	(U)

4. AUTHOR(S)

Kerry Trentelman

5. CORPORATE AUTHOR

DSTO Defence Science and Technology Organisation
PO Box 1500
Edinburgh South Australia 5111 Australia

6a. DSTO NUMBER	6b. AR NUMBER	6c. TYPE OF REPORT	7. DOCUMENT DATE
DSTO-TR-2301	AR-014-554	Technical Report	June 2009

8. FILE NUMBER	9. TASK NUMBER	10. TASK SPONSOR	11. NO. OF PAGES	12. NO. OF REFERENCES
2009/1016220	NS 07/021	Executive Director CTSTC	82	56

13. URL on the World Wide Web

http://www.dsto.defence.gov.au/corporate/reports/DSTO-TR-2301.pdf

14. RELEASE AUTHORITY

Chief,
Command, Control, Communications and Intelligence Division

15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT

Approved for public release

OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE, PO BOX 1500, EDINBURGH, SA 5111

16. DELIBERATE ANNOUNCEMENT

No Limitations

17. CITATION IN OTHER DOCUMENTS Yes

18. DSTO RESEARCH LIBRARY THESAURUS http://web-vic.dsto.defence.gov.au/workareas/library/resources/dsto_thesaurus.shtml

Natural language processing, Intelligence analysis, Reasoning, Logic

19. ABSTRACT

This report describes the theoretical underpinnings of the PENG system. Designed by Rolf Schwitter, Marc Tilbrook, et al. at the Centre for Language Technology at Macquarie University, the system incorporates a text editor where authors write text in a controlled language called PENG. A controlled language processor translates PENG text to first-order logic via a discourse representation structure. The resultant logical theory can then be checked for consistency and informativity, and may also be used for question-answering by third-party reasoning services.